ENABLING COLLABORATIVE, EFFICIENT, AND SCALABLE RESOURCE MANAGEMENT FOR HETEROGENEOUS EDGE COMPUTING ARCHITECTURES

by

Ismet Dagli

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science).

Golden, Colorado

Date ______________________

Signed: ______________________

Ismet Dagli

Signed: ______________________

Dr. Mehmet E. Belviranli
Thesis Advisor

Golden, Colorado

Date ______________________

Signed: ______________________

Dr. R. Iris Bahar
Department Head and Professor
Department of Computer Science

ABSTRACT

Edge computing enables data to be processed closer to where it is generated rather than relying on distant cloud data centers, reducing latency and bandwidth usage. Modern mobile, wearable, and autonomous systems rely on edge intelligence to execute tasks in real time. Consequently, heterogeneous system-on-chips (SoC) on the edge integrate CPUs, GPUs, and multiple domain-specific accelerators (DSA) on a single die. However, the heterogeneous nature of these systems introduces complexities, such as optimal task scheduling across diverse computing resources, minimizing resource contention, and new security vulnerabilities.

In this dissertation, we present novel solutions for energy-aware, latency-constrained, high-throughput, and secure execution of shared-memory heterogeneous SoCs on the edge and their distributed deployments. First, we introduce *AxoNN*, a new energy-aware scheduling scheme that uses different types of DSAs in an SoC. For a given energy budget, *AxoNN* finds the fastest mapping between the layers of a deep neural network (DNN) and the DSAs through constraint-based optimization, achieving time- and energy-prediction accuracies of 98% and 97%, respectively. Following on this, *HaX-CoNN* maximizes the computational throughput of shared-memory SoCs by simultaneously using DSAs in the system via a novel contention-aware runtime. *HaX-CoNN* models per-layer execution times, shared-memory contention, and inter-DSA transitions, and finds optimal schedules for DNNs using SAT solvers, improving latency and throughput by 32% and 29%. Expanding to distributed deployments, we introduce *HARNESS*, a holistic and scalable resource management framework for edge-cloud systems. *HARNESS* relies on a hierarchical graph abstraction that captures varying degrees of heterogeneity within and across computational nodes, and a multi-tiered orchestration mechanism. *HARNESS* improves execution times up to 47% and reduces the prediction error rate from 27.4% to 3.2%. Finally, we unveil a new covert channel attack over shared memory: MC^3, a high-bandwidth covert channel exploiting contention in shared DRAM to communicate between accelerators without requiring a shared last-level cache or privileged access. MC^3 achieves a transmission rate of 6.4 Kbps with an error rate less than 1% and highlights a critical vulnerability in these architectures.

In conclusion, these contributions collectively and holistically provide principled methodologies and tools to optimize performance, energy consumption, scalability, and security of heterogeneous computing systems on the edge.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Completing this PhD would not be possible without the help and support of many people, which I feel immensely grateful for at every step of the journey.

I would like to express my appreciation and gratitude to my advisor, Prof. Mehmet E. Belviranli, whose endless support and timely guidance have shaped every stage of my doctoral journey. From the moment he welcomed me into his research group, he opened doors I never imagined, mentoring me to frame impactful research questions and patiently sharpening my critical-thinking skills. His readiness to meet whenever I hit a roadblock or face stubborn details of the methodologies of my research continually turned challenges into growth opportunities. He always guided me to search for a better version of myself, and his faith in my potential transformed moments of doubt into renewed motivation to evolve into an independent researcher. For his unwavering mentorship and belief in my success, I am profoundly grateful.

My deepest gratitude goes to my wife, E. Beyza Dagli, whose unwavering love and mindfulness sustained me through the innumerable late-night writing sessions that defined my Ph.D. journey. Through every revision and submission cycle, you offered calm reassurance by reminding me to breathe, stretch, and see the bigger picture when the text on the screen began to blur. Your sacrifices, postponed vacations, flexible work hours, and extra household chores, allowed me the privilege of pursuing ambitious goals. This dissertation is as much a reflection of your resilience and partnership as it is of my scholarship, and I cherish that shared achievement with all my heart.

I extend sincere thanks to my thesis committee, Prof. Iris Bahar, Prof. Jamal Rostami, and Prof. Bo Wu. Their valuable feedback and deep questions throughout my journey steered the thesis toward greater clarity and impact. Additionally, I thank to my labmates and colleagues, Justin Davis, Amid Morshedlou, Alexander Cieslewicz, James Crea, Benjamin Wagley, Justin McGowen for collaboration on several research directions and memories of the days we have worked together.

I owe the completion of this thesis to the endless encouragement of my dear friends: Ebubekir Engin, Abdullah Münir Enes Özek, Mustafa Çağlayan, and many others who brightened the stretches of this journey. I thank them for spontaneous talks and the unwavering belief that I could finish what I started.

My sincere gratitude goes to my parents, Emrullah and Nuran Dağlı, and to my brother, Berkant Umut Dağlı. Your quiet sacrifices, constant reassurance, and limitless patience have carried me every step of the way, and I carry your love and support with me into whatever comes next.

Lastly, I acknowledge that this dissertation is supported by NSF 2124010, NSF 2350228, and NIOSH 75D30119C05413, and would not be possible without the support.

CHAPTER 1
INTRODUCTION

Edge computing brings computational capabilities directly to data sources. Emerging mobile, wearable, and autonomous systems rely on edge intelligence for real-time execution of computationally intensive workloads. To satisfy latency and energy requirements in such systems, edge computing architectures are becoming highly heterogeneous with the increasing utilization of domain specific accelerators (DSAs) [1–4] and graphics processing units (GPUs). Over the past decade, computing platforms have embedded both general-purpose CPUs capable of executing any algorithm and DSAs tailored for specific tasks. Each accelerator executes its target kernels (for example transformer attention blocks, sensor-fusion filters, or frame encoders) with significantly greater energy efficiency than a CPU [5–9]. This trend is driven by the need to run applications that span diverse computations for various emerging fields such as deep learning, cyber-physical systems, and virtual reality [10–12]. The latest generations of integrated heterogeneous systems (*i.e.*, NVIDIA's Orin AGX, Apple's M4 and A18, and Qualcomm's Snapdragon X) have brought this degree of architectural heterogeneity onto a single die, integrating diverse processing capabilities. However, consolidating numerous processing units (PUs) introduces several challenges in utilizing them effectively and efficiently for modern workloads. *Addressing those challenges and achieving truly collaborative execution on heterogeneous systems are the primary goals of this thesis.*

As demand for on-device computation continues to grow, the integration of increasingly sophisticated DSAs embedded into *Heterogeneous System-on-Chips (SoCs)* becomes more common [13–15]. The architectural design decisions, such as varying computation/power characteristics among DSAs and CPUs, greatly improve throughput and energy efficiency. This integrated architectural approach enables *collaborative* execution by dynamically mapping each task to the most appropriate PU. Furthermore, system utilization can be improved through *concurrent execution* of tasks across distinct PUs.

A prevalent architectural feature of such heterogeneous systems is a shared memory subsystem, which all DSAs and CPUs can directly access and utilize [16–18]. Shared memory eliminates the overhead associated with data transfer management between the CPU and individual private memories of DSAs. More importantly, direct access to the shared memory by every DSA in the system enables more flexible assignment of workload tasks across available DSAs. This flexibility is fundamental to enabling collaborative execution in shared memory heterogeneous system architectures. Tasks within a workload can be strategically distributed across different DSAs to leverage their specific advantages (*e.g.*,, optimizing for energy, throughput, or latency). For example, a convolution operation might be mapped to a GPU for peak performance or to a deep-learning accelerator (DLA) for superior energy efficiency.

As the computing capabilities of energy-efficient devices advance rapidly, edge computing is becoming more prevalent [19–22]. Edge and cloud systems have emerged as complementary pillars of modern computing. Together, they shape contemporary data-processing architectures and management strategies. Edge computing addresses the growing demand for real-time data processing, delivering instantaneous responses at the data source. Meanwhile, clouds offer significant computational power, centralizing resources and providing vast storage capacities that underpin complex data operations. While combining edge and cloud resources promises major benefits in many emerging domains (*i.e.*, federated learning [23–25], collaborative autonomous robots [26, 27], and virtual reality (VR) systems [28–30]), it also introduces substantial challenges. Particularly, in these emerging domains, a computing system may consist of multiple edge devices with different computational capabilities and cloud servers. Providing efficient execution becomes more challenging as more edge devices and cloud servers are integrated into the system. Striking a balance between the low-latency, yet resource-constrained, edge computations offered by edge devices with the expansive computational capabilities accompanied by higher communication latency of the cloud resources, while ensuring efficiency and scalability, remains a central research problem.

Application requirements and resource constraints fundamentally guide both design-time architectural decisions and runtime scheduling strategies [31–34]. For instance, cyber-physical systems frequently incorporate resource-constrained architectures consisting of diverse PUs, such as GPUs, DLAs, and tensor processing units (TPU), connected to the sensors interacting with the physical world. Despite having a variety of accelerators available, systems with time-critical requirements or strict physical constraints (*i.e.*, energy) necessitate careful management of resource utilization across specific PUs (*i.e.*, CPUs or DSAs). Exceeding the allocated time or energy budget can cause critical system failures. Similarly, VR headsets equipped with heterogeneous SoCs must constrain computational latency to meet specified quality-of-service (QoS) thresholds. Concurrently, servers serving to multiple edge devices must maintain consistent latency within the same threshold for each device. Such constraints necessitate in-depth performance analysis throughout the design phase and runtime operations. Operational decisions in such systems must also respect the platform's resource constraints, such as VR headset's power and energy constraints. While the design of such systems aims to adhere to these constraints, the runtime must be able to adapt to dynamic system changes (*e.g.*, the addition of a new edge device). This requires real-time adaptation and continuous management of the system in response to dynamic hardware configurations, computational demands, and energy limitations.

1.1 Targeted Research Challenges

In general, efficient, collaborative, and secure execution of modern workloads on heterogeneous edge devices necessitates a comprehensive understanding of performance capabilities, energy consumption, and security vulnerabilities at both the SoC and system levels. This thesis tackles the following critical challenges.

1.1.1 Minimizing Energy Consumption

Energy is a first-order constraint in autonomous robots, aerial drones, smart cameras, and battery-powered VR headsets. The deployment of machine learning tasks, particularly deep neural networks (DNNs), incurs significant energy consumption. Thus, DSAs have become standard because they deliver an order-of-magnitude better joules-per-inference than a GPU executing the same kernel [35, 36]. Depending on the current energy budget of the system, the scheduler can choose between two strategies: (i) assigning a task (*e.g.*, the next DNN layer) to the single most energy-efficient PU, or (ii) distributing tasks across multiple DSAs to achieve a performance goal while optimizing for overall SoC energy consumption. Given that these systems aim to execute operations in real time with energy as a dynamic constraint, the methodologies must be inherently adaptable to runtime conditions. Energy-conservative DSAs generally outperform commonly employed energy-saving techniques, such as dynamic voltage and frequency scaling (DVFS), since such DSAs are architected to specifically accelerate a type of workload (*e.g.*, layers in a DNN) in a highly optimized manner [37]. Mapping each task in a workload to the DSA that can execute it most efficiently can yield optimal energy savings, though sometimes at the cost of slower execution time.

1.1.2 Minimizing Computational Latency

Reducing computational latency requires pinpointing performance bottlenecks at both the SoC level and the system level. Every PU in a heterogeneous system exhibits unique computational characteristics for different workloads. For example, an autonomous platform that runs several concurrent tasks (such as perception tasks, localization, mapping, and planning) must carefully map each task to an appropriate DSA. However, parallel execution of multiple tasks can trigger resource contention within the device, especially within the shared memory subsystem, thereby introducing additional latency. Consequently, assigning tasks based solely on standalone execution-time profiling produces inaccurate performance predictions. Exhaustively exploring every possible task-to-DSA mapping is infeasible because of the vast design space, and many DSAs, by design, expose only limited profiling information regarding their capabilities and the granularity of available profiling data. Numerous black-box DSAs are released with scant public data about their performance or architecture. A practical framework must therefore accommodate the black-box nature of these emerging accelerators to deliver precise system-level performance predictions and to guide the efficient allocation of resources.

1.1.3 Improving Total System Throughput

As the deployments expand to include many devices, improving total system performance requires detailed analysis and utilization of resource use at both the SoC and the system levels. At the SoC level, throughput is governed by the capabilities of embedded PUs (CPUs and DSAs). A common strategy for boosting performance is to run each task on whichever PU completes it fastest; however, in a heterogeneous SoC, this approach might leave powerful units idle while they wait for tasks executing on slower units to finish, thereby limiting overall speed-up. To achieve higher throughput, a more comprehensive mapping approach spanning multiple applications and PUs is necessary, while considering the shared resource slowdown (*e.g.*, shared memory contention). Similar issues arise in larger systems, where server-side resource allocation mechanisms (*e.g.*, cloud) must consider edge-device performance, and vice versa. Computing resources in each tier, such as PUs within edge devices and high-performance servers in datacenters, are handled in isolation due to scalability and resource segregation. This practice results in task mappings limited to only a subset of all available PUs, preventing efficient overall utilization of the system. Maximizing the overall throughput in such platforms requires a holistic view of the entire system's performance while still respecting the isolation and resource segregation. New solutions should address the issues of PU heterogeneity, edge and server-side scalability, runtime adaptability, and shared resource contention.

1.1.4 Identifying Security Vulnerabilities of Shared Memory Design

In shared memory SoCs, while system-wide shared memory enables a convenient and cost-effective mechanism for making data accessible across multiple PUs, such as CPU cores and domain-specific accelerators, this architectural feature can also create new avenues for an attack. Due to the diverse computational characteristics of the PUs they embed, shared memory SoCs often do not employ a shared last-level cache (LLC). From a security perspective, covert channel attacks have been widely demonstrated with high transmission rates that take advantage of CPU caches (*i.e.*, L2 and LLC); however, achieving high capacity covert channels over shared memory designs has been more challenging. Existing trusted-execution environments (TEEs) and related countermeasures offer limited protection against covert channel attacks that exploit the shared-memory subsystem. Although covert-channel attacks have already been studied in shared-memory contexts, high-throughput communication has previously been practical only when attackers could leverage an LLC or hold privileged or physical access to the memory system.

Addressing these vulnerabilities, therefore, remains an essential component of a secure heterogeneous platform.

1.2 Proposed Solutions

Modern heterogeneous edge platforms face four orthogonal yet tightly coupled challenges listed above. In this thesis, we develop a set of strategies to jointly resolve the barriers to predictable, efficient, and secure heterogeneous execution.

(i) Collaborative multi-accelerator execution for energy-capped systems: To limit the energy cost of workloads while preserving responsiveness, we characterize the execution time and energy for all tasks in the workload on each on-chip accelerator and measure the cost of switching execution between accelerators, if needed. These measurements are embedded in a constrained-based objective optimization formulation that, for any user-supplied energy target, produces a layer-to-accelerator mapping with the lowest achievable end-to-end latency. We demonstrate the effectiveness of our approach on DNNs.

(ii) Contention-aware, fine-granular concurrent execution on multi-accelerator SoCs: For workloads that need to run simultaneous tasks, we first decouple intrinsic per-task latency from slowdown under contention, then express the joint task-mapping problem as a satisfiability-modulo-theory (SMT) instance. The solver selects layer groups so that (i) neither accelerator is idle, (ii) the number of inter-accelerator transitions is bounded, and (iii) concurrent tasks respect the available DRAM bandwidth. A lightweight runtime re-optimizes the schedule whenever the control-flow graph of the workload changes.

(iii) Hierarchical edge–cloud orchestration that preserves QoS while respecting isolation and segregation between computational nodes: We model the entire continuum as a multi-layer hardware graph in which the vertices represent PUs and memory, and the edges represent interconnects. A modular *Traverser* predicts execution time and interference along candidate paths, while a hierarchy of decentralized *Orchestrators* negotiates task placements that satisfy QoS constraints without requiring any tier to hold a global performance model.

(iv) Identifying security vulnerabilities on shared-memory SoCs: We construct a memory-contention-based covert communication channel attack that does not require a shared last-level cache design or elevated privileges. By crafting non-temporal copy kernels running on the CPU and GPU and synchronizing their activity between both PUs, the transmitter imposes a deterministic bandwidth signature through DRAM access latency patterns. We further analyze the impact of buffer size and evaluate the trade-off between accuracy and channel bandwidth capacity.

1.3 Key Contributions

This thesis advances heterogeneous computing in edge platforms in several dimensions, such as resource management with formal theory, practical multi-accelerator scheduling algorithms, cross-tier and generalized system orchestration through edge and cloud collaborative platforms, and architectural security. Thus, this thesis delivers a cohesive framework that future researchers and practitioners can build upon and utilize.

Increased performance and energy-efficiency through the use of different types of accelerators: The increasing demands of modern computing, particularly driven by complex AI workloads, have made the traditional approach of simply increasing microprocessor clock speeds or core counts insufficient. While raw computational power continues to advance, many workloads necessitate a more nuanced strategy. To meet stringent performance and energy budgets in mobile and autonomous systems, the efficient utilization of heterogeneous accelerators within SoCs has become paramount. We empirically demonstrate that sustained performance and energy improvements must come from simultaneously harnessing the diverse accelerators already present on commodity SoCs. We propose a methodological and generalizable approach to increase performance and energy efficiency for any mobile or autonomous system built around multi-accelerator SoCs. The implications of this work are substantial, impacting billions of devices by enabling them to execute complex tasks more effectively and sustainably, thereby extending operational lifetimes and expanding capabilities in resource-constrained environments.

Constraint-based near-optimal task scheduling through solvers for multi-accelerator platforms: A fundamental contribution of this research lies in effective utilization of formal methods (*i.e.*, Integer Linear Programming (ILP) and SMT) in solving complex, multi-objective scheduling problems on heterogeneous SoCs. We demonstrate the capacity to satisfy the system constraints, such as energy consumption thresholds and resource contention limits, and to exploit the distinct capabilities of diverse accelerators to derive near-optimal, verifiable scheduling solutions. This is achieved by formulating energy-aware scheduling for individual tasks and contention-aware scheduling for concurrent workloads as constraint-solving problems, particularly for applications characterizable by tasks and a control flow graph (CFG). Then, we specifically leverage the advantages of utilizing heterogeneous accelerators across available tasks. Our workflow offers a reusable blueprint for future architectures integrating multiple domain-specific accelerators, thereby shifting the community's emphasis from ad-hoc heuristics to formally verifiable optimizations.

A novel hardware representation for end-to-end scalable resource management: We design the first hardware-graph-based abstraction that is *multi-layered*, capturing processors, memories, and interconnects within and between each computational node, and *hierarchical*, propagating interference information across nodes. When paired with decentralized orchestrators, the abstraction proves that accurate performance prediction and QoS-driven placement are achievable without a monolithic controller, setting a template for future work in several domains, including AR/VR platforms with edge devices and cloud services, federated learning clusters, and autonomous vehicle swarms.

Security implications of shared memory design in multi-accelerator platforms: We expose a critical vulnerability in modern heterogeneous SoCs, highlighting the inadequacy of current security defenses and necessitating new DRAM arbitration countermeasures. We reveal that high-bandwidth covert channels can be built solely from DRAM contention with no privileged access and without targeting cache in the memory hierarchy. The attack broadens the scope of architectural security research to include memory-centric side channels in GPUs and additional domain specific accelerators (DLA, TPU, etc.), which had previously been assumed safe. Our demonstration exposes a covert channel of kilobit per second that requires neither caches nor privileged code, thus shifting the attention of the security community to widely used shared DRAM SoCs without cache.

1.4 Dissertation Overview

This dissertation is organized as follows. In Chapter 2, we list several related works for energy-efficient and high-throughput execution of DNN inference, edge and cloud collaborative executions, and existing security vulnerabilities for shared memory architectures. In Chapter 3, we present AxoNN [38], a collaborative and energy-aware multi-DSA execution scheme for DNN inference on heterogeneous SoCs. In Chapter 4, we introduce HaX-CoNN [39], a throughput efficient, multi-accelerator, and contention-aware runtime for concurrently executing DNNs on shared memory SoCs. In Chapter 5, we build *HARNESS* [40], a holistic approach that captures the diverse computational characteristics of edge-cloud systems with arbitrary topologies and efficiently manages computational resources with the whole continuum in scope. In Chapter 6, we unveil MC^3 [41], a new memory-contention-based covert communication attack that specifically targets shared system memory in mobile SoCs. In Chapter 7, we propose future work ideas on efficient scheduling for emerging workloads on SoCs, context-aware resource management, and countermeasures for covert channel attacks and conclude the thesis.

CHAPTER 2
RELATED WORK

In this chapter, we review state-of-the-art for energy-aware and throughput-optimized execution on shared-memory SoCs, resource management in edge/cloud systems, and security vulnerabilities in shared DRAM architectures.

2.1 Energy-aware Heterogeneous Execution on Shared Memory SoCs

Taking advantage of the different types of DSAs for energy-efficient execution of emerging workloads, such as DNN inference, requires scheduling strategies that balance performance against tight energy budgets while respecting inter-accelerator data-movement costs and memory contention. In the following subsections, we review the state-of-the-art in energy-aware DNN pipeline partitioning, DVFS-driven layer mapping, and analytically guided trade-off models that have been proposed to orchestrate such multi-DSA platforms, setting the stage for the design space our work targets.

Pipeline parallelism: The data to be processed on DNNs can be distributed in small batches by creating pipeline parallelism for multiple DSAs. GPipe [42] and PipeDream [43] split tasks between multi-DSAs using pipelines and considering the transition time between accelerators for deep learning (DL) training. HetPipe [44] considers the first step of heterogeneity and sets up multi-GPU clusters to apply the idea of pipeline parallelism by maximizing utilization. However, none of the aforementioned works takes energy into account.

Narayanan et al. [45] propose round-robin scheduling for a target-latency deadline for DL workloads. Shamsa et al. [46] prioritize resource management over goals by considering dynamic changes. PCCS [47] presents an empirical model that formulates shared memory contention between multiple DSAs. However, these studies do not consider energy as a target or a metric.

Pipelining in DNN inference [48] is applied by distributing layers between CPU and GPU to maximize system throughput and synchronous data through cache-coherent interconnects. Kang et al. [49] optimize a single DL application response time using DVFS technique by finding the Pareto-optimal scheduling. Jeong et al. [50] offer a parallelization methodology for DNN inference workloads to maximize throughput by leveraging TensorRT's GPU and DLA. However, none of these works considers using multiple accelerators for DL tasks to find a trade-off between energy and latency. Minakova et al. [51] present a mechanism for the execution of convolutional neural network (CNN) inference on MPSoC integrated CPUs-GPUs by using task-level (pipeline) and data-level parallelism. Soomro et al. [52] uses pipeline to maximize throughput with an online-tuning approach to decide the mapping of tasks to DSAs.

Energy-performance trade-offs: MEPHESTO [53] first characterizes the workloads on different DSAs and then models an energy-performance trade-off by collocating kernels and considering the memory contention on heterogeneous shared memory systems. Their model considers the ordering and placement of any given tasks between heterogeneous DSAs by using a dynamic programming algorithm. However, this study does not take either the dependencies between OPs or the transition costs into account and is therefore not suitable for DNN inference. Barik et al. [54] propose a mapping algorithm between CPU and GPU by characterizing the power consumption of data-parallel specific workloads. Tzilis et al. [55] propose an online profiling model to estimate power consumption and performance under DVFS configuration for a given application. The aforementioned works do not consider challenges in executing DL applications on multiple DSAs.

Concurrent DNN execution: Several studies [56–61] propose scheduling techniques for the concurrent execution of multiple DNNs. Two of them focus on multi-DNN inference on SoCs: Herald [56] introduces a mapper to optimize hardware resource utilization across accelerators such as NVDLA [5] and Shi-diannao [62] whereas H2H [57] improves Herald by considering inter-accelerator transition costs. OmniBoost [59] uses Monte Carlo tree search by exhaustively profiling layers on CPU and GPU and generates optimal, yet static schedules.

Multi-accelerator scheduling: Scheduling for systems with more than one type of accelerator has recently been targeted by many studies [63–69]. Among the most relevant, Gamma [63] and Kang et al. [64] build genetic algorithms to utilize multiple accelerators for a single DNN execution while Wu et al. [65] and Mensa [66] target unique hardware for edge devices. None of these studies address contention and balancing issues with multi-DNN execution. DNN training for large-scale systems [70–73], on the other hand, is outside the scope of this work.

Shared memory contention: None of the studies mentioned so far addresses shared memory contention. MoCA [58] designs a multitenant DSA architecture with dynamic memory resource management. FAST [74] uses an integer linear programming based operator fusion technique to remedy the memory bottlenecks whereas ParDNN [75] partitions DNNs under a memory limit. However, these approaches are not adaptable to off-the-shelf multi-DSA shared memory SoCs.

2.2 Edge and Cloud Collaborative Execution

We review various performance modeling and resource management approaches for collaborative execution of applications on diversely scaled edge cloud systems (DECS).

Performance modeling for diverse heterogeneity: Understanding and modeling the performance of DSAs in heterogeneous systems have been widely studied [11, 76–82]. In these systems, interference factors that adversely affect performance include shared caches [83, 84], CPU sharing [85–87], GPU multi-tenancy [88–90], and multi-tasked DSAs [58, 91, 92]. However, all these works study a shallow, *i.e.*, *single-tier*, case of heterogeneity: They assume either there is a single type of DSA or a single flat layer of interaction between DSAs. In DECSs, however, computational nodes and the DSAs are connected in arbitrary topologies with multi-tiered hierarchies and they are often abstracted behind segregated clusters. Existing approaches are not flexible and generalizable enough to capture multi-tiered interactions between diversely heterogeneous PUs in a DECS.

Shared resource slowdown: Recent studies [13, 39, 93, 94] identified that tasks running concurrently on the PUs of shared-memory SoCs are subject to significant slowdowns. At the cloud level, multi-tenancy is often unavoidable [95, 96] since each server serves requests coming from multiple edge devices. These two types of slowdown are often more severe in performance-limited edge devices than more commonly studied high-performance systems in the cloud [83, 84]. Therefore, if resource management mechanisms in DECSs do not account for slowdowns at different tiers of computation, predicted performance will be inaccurate, leading to missed QoS targets and oversubscribed resources.

HW and system representation schemes: Graphs have been commonly used to represent system topologies in traditional large-scale systems [97–99], cloud providers [100], network-on-chip modeling [101, 102] and electronic design automation [103]. Among the most notable, *Hwloc* [3, 104, 105], an approach focused on high-performance computing (HPC) systems, uses a tree data structure to represent the underlying HW. However, when used for DECSs, these approaches (i) are incapable of adaptive detection of shared resource contention and propagation of the slowdown across different layers, (ii) do not support segregated resource clusters and abstractions, and (iii) are not versatile enough to support dynamic changes to the HW topology.

Scalable and adaptive resource allocation and management: Many task-based execution frameworks [106–112] have been proposed for heterogeneous HPC systems. Interference-aware resource management within GPUs [113] and across large-scale clusters [110, 114–117] is also widely studied for server-based systems [118, 119]. Most recently, multi-accelerator collaborative execution in embedded system SoCs [56, 66, 120, 121] attracted attention. A few studies [28, 76, 122–124] focused on task mapping in edge-cloud platforms, however, they are either hand-tuned for specific application and HW, or they ignore diverse heterogeneity.

2.3 Security Concerns and Vulnerabilities for Covert Channel Attacks

Over the years, security researchers have shown that having a shared hardware component with a predictable performance slowdown leaves a unique fingerprint. Using this fingerprint, various attacks have exploited cache [125], memory [126], storage [127], temperature [128], and power [129]. Covert channel communication attacks targeting vulnerabilities in the memory subsystem can be categorized into three: (1) *Cache-based, high-throughput attacks* which leverage the LLC between CPU cores [125, 130–133], between multiple GPUs [134] and between a CPU and a GPU [135]. (2) *Low-throughput attacks targeting directly the DRAM* rely on memory performance attacks [136], memory deduplication [137, 138], bus snooping [126] and monitoring of DRAM power consumption [139]. (3) *Attacks requiring elevated privileges or hardware access* [136, 140–142]. None of these studies have shown how to build a fast, memory-contention-based covert channel without the need for privileged access (check Section 6.3.1).

Denial of service (DoS) attacks [130, 143] exhaust the memory subsystem and greatly increase memory access latency. Commonly implemented memory controller (MC) scheduling policies, such as fairness control [144] and adaptive scheduler [145], are typically designed to maximize system performance. Although MC schedulers that prioritize security [146, 147] mitigate memory performance attacks with limited overhead, they cannot fully eliminate the threat. Our work, however, focuses on covert-channel communication, which demands far higher precision than DoS attacks require (check Chapter 6).

Additional studies have shown how to create architectural covert channels in the cloud [126, 130], HPC servers [134], and desktops [148, 149]. Similar studies also used temperature [128, 129] and power [129, 150] as a covert communication channel. Our work focuses on the vulnerabilities stemming from shared memory use.

CHAPTER 3
AXONN: ENERGY-AWARE EXECUTION OF NEURAL NETWORK INFERENCE ON MULTI-ACCELERATOR HETEROGENEOUS SOCS

The energy and latency demands of critical workload execution, such as object detection, in embedded systems vary based on the physical system state and other external factors. The execution flow of the critical workloads can be adjusted to span into multiple accelerators so that the trade-off between performance and energy fits to the dynamically changing physical factors. In this chapter, we will present our framework AxoNN. AxoNN is an energy-aware DNN inference scheduling framework for multiple accelerators under an SoC by enabling an energy-performance trade-off by distributing layers in a DNN between a performance- and a power-efficient accelerator.

3.1 Introduction

Computing devices are becoming highly heterogeneous with the increased utilization of domain specific accelerators (DSAs), each of which is optimized to perform a specific type of operation. This trend is fueled by the need of running applications that span a diverse set of computations for emerging fields such as artificial intelligence, machine learning, and autonomous systems. The latest generation of SoCs —such as NVIDIA's Xavier and Orin architectures, Apple's M1 and A15 Bionic chips, and Qualcomm's Snapdragon 8 SoCs— have dramatically increased the degree of architectural heterogeneity within the same die. In such systems, dozens of DSAs with diverse instruction set architectures (ISAs) work together to accelerate operations (*i.e.*, kernels or tasks in an application) that belong to emerging application domains.

In diversely heterogeneous SoCs, an operation (OP) can often be accelerated via different DSAs with varying performance, energy, and latency characteristics. For example, a convolution OP can be set to run on the CPU, GPU, deep learning accelerator (DLA) and programmable vision accelerator (PVA). The DSA that would provide the near-optimal execution time and/or energy efficiency for an OP depends both on the DSA capabilities, and on the properties of the operation, such as matrix size and filter dimensions. Depending on the dynamic requirements of the system (*e.g.*, high throughput, low energy), runtime parameters of an OP (*e.g.*, number of objects, image size), and availability of DSAs, the programmer (or the system scheduler) may choose to map different OPs to different DSAs throughout the execution of an application.

An emerging architectural feature of such heterogeneous SoCs is a *shared system memory* that all DSAs and CPU can directly access and utilize. While this design choice is primarily motivated by the goal of reducing chip area and production costs, it also helps in eliminating additional data transfer overhead between the host and the device [151]. Having shared memory directly accessible by every DSA in the system enables assigning OPs in a workload to the DSAs more flexibly. This flexibility also enables *collaborative execution* in shared-memory heterogeneous system architectures (SM-HSA), where OPs in a workload can be executed on different DSAs [152] to exploit the varying benefits (*i.e.*, energy, throughput, latency, etc.) that different types of DSAs can optimally provide—*e.g.*, a convolution operation can be accelerated by a GPU for high performance, or by a DLA for better energy efficiency. In such SM-HSAs, near-optimal utilization of the system resources heavily relies on carefully assigning the OPs to the available DSAs based on the target performance and power goals of a given scenario [53].

While the most [44, 153–157] focus on improving the total throughput by using multiple DSAs concurrently, only a few [54, 55] have investigated the performance-energy trade-off in limited aspects. Besides, a limited number of studies [50, 158, 159] explore the benefits of using different types of DSAs collaboratively for the same application. To the best of our knowledge, none of the existing studies are able to address all of the following challenges altogether for multi-DSA collaborative execution:

- Holistic modeling of multi-accelerator execution that takes both the execution and data-transfer costs between different type of DSAs into consideration.
- Tunable objective for power consumption which can be targeted while finding schedules with optimal execution time.
- Generalized layer-wise characterization methodology for finding performance and energy costs of neural network inference on multi-DSA systems.

In this study, we propose an energy-aware multi-DSA execution scheme for DNN inference on heterogeneous SoCs. Our proposed scheme, *AxoNN*, uniquely enables setting an energy consumption target (ECT) and finds a NN-layers-to-DSA mapping that minimizes the total execution time under a given ECT. *AxoNN* utilizes a novel inter-accelerator transitional cost model to integrate the penalty of switching between DSAs into the cost function. Our scheme characterizes each layer in the network for each target DSA, and explores multiple transitions between layers to find schedules that satisfy the given ECT. We represent the scheduling problem as a constrained-objective optimization problem.

3.1.1 Contributions

Our paper makes the following contributions:

- We present *AxoNN*, a multi-accelerator execution scheme for diversely heterogeneous SoCs, which finds schedules with near-minimal execution time for a given ECT.
- We propose a novel, empirical model-creation technique to represent the cost of inter-DSA transitions on a shared-memory heterogeneous SoC.

- We build cost models for estimating energy and execution times which uniquely take transition times and hardware-pipelined accelerator architectures into account.
- We evaluate *AxoNN* on the NVIDIA Xavier AGX SoC by using its embedded GPU and DLA. Our results show that our methodology can find near-optimal schedules with one or two inter-DSA transitions within up to 98% and 97% time and energy prediction accuracy, respectively, while staying under the given ECT.

The remainder of this chapter is organized as follows: We first give a motivational scenario and challenges in Section 2. Then, we demonstrate the communication and computation characterization for multi-accelerator execution in Section 3. We list the necessary components for modeling the methodology in Section 4 and present the methodology in Section 5. We present the evaluation of AxoNN in Section 6.

3.2 Motivational Study and Challenges

Motivation: Collaborative execution of popular workloads, such as neural network (NN) inference, on different types of DSAs is a relatively new and unexplored scheme which has the potential to provide unique benefits for budgeted execution scenarios. To demonstrate the feasibility of collaborative execution for achieving different performance and energy goals on a heterogeneous platform, we conduct an exploratory experiment, which is shown in Figure 3.1. In this experiment, we map the layers of the VGG-19 [160] network to the GPU and the DLA of NVIDIA's Xavier AGX SoC in three different ways. The left-most and right-most columns in the figure show where all layers are executed, either on the GPU or the DLA, respectively. The middle column illustrates a *collaborative* execution where the first n layers are run on the GPU, and the remaining m layers on the DLA. The total execution time and energy consumed are given under each column. Experimental results show that running all layers on the GPU results in the fastest execution time, whereas running all layers on the DLA is the most energy-efficient. On the other hand, the collaborative execution scheme shown in the middle results in a trade-off between execution time and energy, as more layers of the network are executed on the DLA.

A hybrid (i.e., GPU+DLA) execution scheme could be more feasible in real-life scenarios, when there is an energy constraint in the system. For example, when an autonomous aerial drone is running low on battery, scheduling of the DNN layers to DSAs can be adjusted at the expense of a higher execution time (i.e., latency), hence resulting in a lower images/second detected by the NN. For the scenario given in Figure 3.1, if the remaining energy budget per image detected is less than 205 milijoules, (total energy/image needed for GPU-only execution), but more than 139 milijoules, rather than running the entire DNN on the DLA, choosing the GPU+DLA hybrid schedule in the middle will result in a more feasible operation. *The drone will still be able to complete its flight, but will be more responsive to the* *surrounding objects*, thanks to the lower latency (12.4 ms per image) achieved by a GPU+DLA hybrid execution against a DLA-only execution (22.6 ms per image).

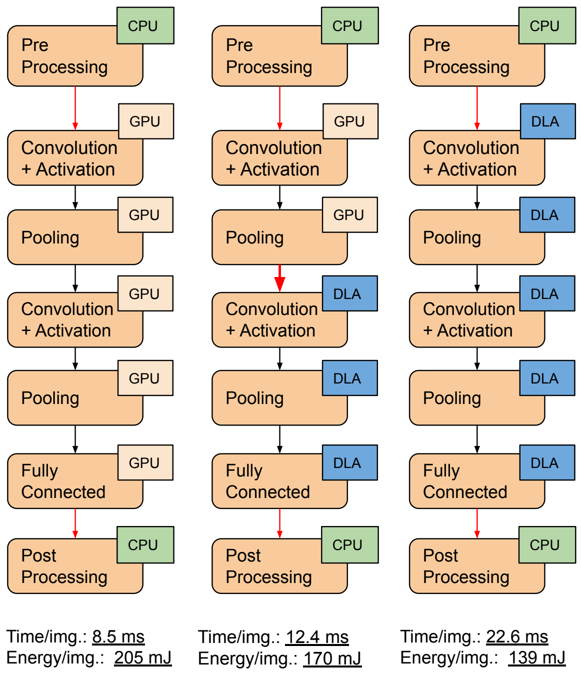


Figure 3.1 Simplified layer mappings for VGG-19 when executed with TensorRT on Xavier AGX: Leftmost and rightmost control flow graphs (CFG) show the traditional methods of executing the DNN on a single type of DSA. The multi-accelerator execution shown in the middle employs a transition between DSAs in the execution flow to produce a schedule with a trade-off between latency and energy.

Challenges: We list some existing strategies of optimizations for efficient execution and identify the following challenges to keep the optimization.

1) Grouping layers: Operator fusion has become a commonly applied optimization by popular frameworks, such as TensorRT [161] and TVM [162], that minimizes the cache misses between OPs and eliminates the duplicate OPs. Breaking potentially-fusible operations will increase execution time and, as a result, energy usage.

2) Lack of flexibility in layer-to-DSA assignment: Each DSA has a different set of restrictions in terms of their capabilities for running different OPs. For example, even though layer activation functions are considered as separate layers on TensorRT layer set, TensorRT scheduler does not allow assigning activation layers and other layers separately between DSAs. NVIDIA's DLA has additional restrictions on layer parameters and batch sizes. Moreover, TensorRT does not allow to transitions from DLA to GPU after certain (*e.g.*, Eltwise) layers . Such limitations force some layers to fall back to the GPU, even though they are supposed to execute on DLA. Therefore, the flexibility on the potential inter-DSA transition points is restricted, and depends on the architecture of a NN, and on DSAs.

3) Profiling: Some highly-specialized accelerators —such as DLAs— run the consecutively-assigned layers as a single black box and do not allow internal profiling of execution times layer-by-layer. This limitation makes empirical modeling more challenging, since it presents an obstacle to fine-grained performance characterization.

3.3 Characterization for Multi-accelerator DNN Inference on Diversely Heterogeneous SoCs

In DNN inference, finding the desired trade-off requires a careful distribution of layers onto accelerators. However, using multiple DSAs to maximize the system's utilization while staying under resource constraints, such as ECT, requires characterizing the computation and communication cost of tasks.

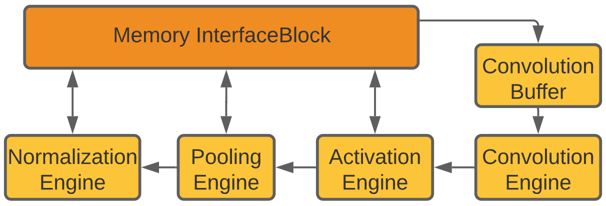


Figure 3.2 A simplified internal block diagram for NVIDIA's DLA.

3.3.1 Inter-DSA Transition Overhead

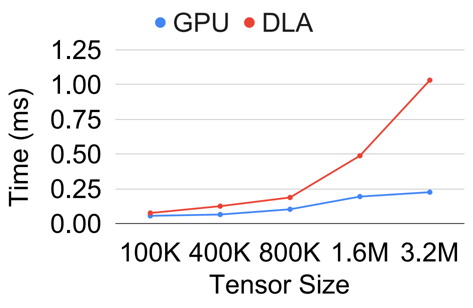
On shared-memory SoCs, caches are often private to DSAs, due to complexity of cache coherency across diversely heterogeneous processing units. When the execution flow switches from one DSA to another on a shared memory system, which we call as *transition points*, the transient data present in private caches or buffers of DSAs needs to be written back to the shared memory. Such additional memory read/write operations need to be considered as transition overhead and added to the total execution time. The size of the memory pages being written or read is the primary factor determining the magnitude of this transition overhead. In most NNs, the size of the input and output data that each layer consumes and produces often changes after each layer. Moreover, the internal memory hierarchy of each DSA affects the transition overhead differently, even though the amount of data being read or written by two DSAs is the same [50]. Therefore, modeling the cost of inter-DSA transitions requires a careful consideration of the data size, layer type and the DSA that the transition originates from and/or arrives into.

3.3.2 Execution Time Characterization

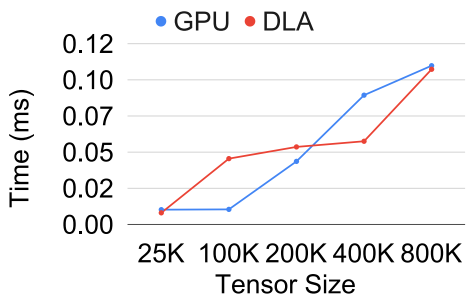
While the execution time of a specific DNN layer on a given DSA primarily depends on the layer type and the input data size, the location of the data before the layer is also an important factor. Therefore, we model layer execution time and inter-DSA transition time separately. The cold cache misses issued by DSAs as they begin executing a layer after a transition requires a warm-up period for layer-by-layer characterization. Another important factor affecting the execution time is the existence of internal hardware pipelines. As shown in Figure 3.2, NVIDIA's DLA architecture embeds a pipeline of internal engines for common layers, such as convolution, activation, and pooling, in the order that these layers often appear in NNs. The data between the engines are often forwarded with direct data buses, and separate characterization of such layers may result in incorrectly estimated layer execution times. For example, since the pooling layer reduces the amount of data being passed to the next layer, measuring the execution time of the convolution and activation layers separately from the pooling layer will result in a longer execution time than the case where these three layers are profiled together. Therefore, layer-characterization needs to take such HW behavior into account for HW-pipelined DSAs.

3.4 Modeling Methodology

Taking into account the challenges and constraints, this section explains the methodology we utilize to build our empirical model.



(a) Convolution layer



(b) Pooling layer

Figure 3.3 Empirical models for Convolution and Pooling layers' out-transition times after the layers are run on GPU and DLA. X axis indicates the tensor sizes.

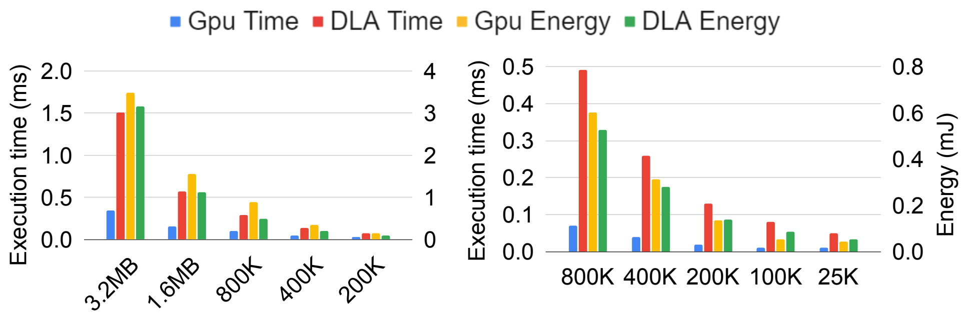
3.4.1 Modeling Inter-DSA Transition Cost

For the size of data needed to perform read/write operations, the transition overhead will increase as the size of the data increases. The X-axes in Figure 3.3.a and Figure 3.3.b represent the tensor sizes, *i.e.*, the size of the output produced after convolution/pooling layers. The Y-axis represents the transition cost in milliseconds if any transition is applied after the convolution/pooling layers. For example, performing the transition operation for the convolution layer with a data size of 3MB can result in 14x more time overhead compared to a data size of 100KB on the DLA. Thus, it is clear that the transition overhead decreases as the data movement decreases on both of the devices in our experiments. Besides, DLA has a second private buffer specifically for convolution operations, which directly affects DLA's behavior for different data sizes. Since the buffer has a relatively more limited size, larger data necessitates data movement from afar rather than a private buffer.

3.4.2 Energy and Performance Characterization

Based on architectural restrictions, we check whether a layer can run on all DSAs, or can be marked as a transition layer by using *canRunOnDLA* and *markOutput* TensorRT API calls. For example, a ReLU activation layer cannot run by itself on the DLA, but merging a ReLU activation layer with a convolution layer enables running both of them on the DLA. All of these values are obtained via offline profiler *IProfiler*, using an API call on TensorRT, to obtain execution time and energy consumption on GPU and DLA.

We analyze the execution behavior of the DSAs for different OPs by considering layer types and OP complexities, and measure the NN's resource utilization layer-by-layer. In Figure 3.4(a-b), we measure execution time and energy consumption of different layers on VGG-19 by using GPU and DLA separately. The left vertical axis shows the execution time, whereas the right vertical axis represents the energy consumption. Depending on the data size, convolution OPs on GPU are 3x to 4.5x faster than on DLA, whereas pooling OPs on GPU are 3x to 7x faster than on DLA. For the energy comparison, convolution OPs on GPU consumes 1.1x to 1.8x more energy than on DLA, whereas the ratio for pooling on GPU over DLA varies from 0.6x to 1.15x.



(a) Convolution (b) Pooling

Figure 3.4 Empirical models for execution times of Convolution and Pooling layers on GPU and DLA for different tensor sizes.

3.5 Multi-accelerator Scheduling via Constraint-based Optimization

This section details how we build our cost functions and encode scheduling as a constraint-based optimization problem. First, we formulate the total execution time using empirical values found for transition and layer-wise execution times as shown in Section 3.4. Then, we assemble a constraint-based optimization problem that minimizes total execution time for a given ECT.

Table 3.1 The notations used in this section.

Notation	Explanation
L_i	ith layer in a given layer set L, $0 \leq i \leq len(L)$
P_j	jth processor in a given processor set P, $0 \leq i \leq len(P)$
TR_i	Boolean variable set if a transition occurrs after L_i, $0 \leq i < len(L)$
$\tau(i, j, OUT)$	Output transition cost inflicted on P_i as the layer L_i transitions to P_j.
$\tau(i, j, IN)$	Input transition cost inflicted on P_j as the later L_i transitions from P_i.
$e(i, j)$	Energy consumption of layer L_i on processor P_j
$t(i, j)$	Execution time of layer L_i on processor P_j
ECT	Energy consumption target
S(L_i)	Scheduling set of layers, 0 < i < len(L)
$T(L, P, S, TR)$	Total time to execute a given layer set L, processor P, and schedule S
$E(L, P, S, TR)$	Energy consumption by a given set of layer L, processor P, and schedule S
$U(Li)$	The sub-unit executing the layer
$\gamma(L_i, s(L_i))$	Amount of time L_i saves by pipelining in its input
NumTransitions	Maximum amount of transitions allowed by user

Table 3.1 lists the components needed to formulate our optimization problem. Layer set L is a NN-specific parameter which includes all layers L_i in the network with their sizes and activation functions. Processor set P represents all available processors and DSAs on the architecture. $\tau(i, j, OUT)$ is used to denote the OUT transition overhead inflicted on P_i as the transient data belonging to recently executed layer are flushed back to the system memory. In this case, P_j is the target DSA of the transition. Similarly, $\tau(i, j, IN)$ represents the IN transition overhead inflicted on P_j due to the cold cache misses caused by the initial memory instructions executed by P_j.

Since each layer can be mapped to a different processor, the final layer-to-processor schedule for a neural network is represented by:

$$S(L_i) = P_j \qquad where \quad 0 < i < m \quad \& \quad 0 < j < n \tag{3.1}$$

Since breaking fused operations and pipelined operations will cost extra time overhead, we consider another feature in our metholodogy, $pipeline(L_i, S(L_i))$ in Eq. 3.2. If the schedule does not prevent any OP from being pipelined, there will be no effect on the time and energy parameters. However, if the sub-unit used by the previous layer is not the same sub-unit on the current layer for the same DSA, it can severely affect execution time and energy results.

$$pipeline(L_i, S(L_i)) = \begin{cases} 0 & \text{if } U(L_i) = U(L_i - 1) \\ \gamma(L_i, s(L_i)) & \text{if } U(L_i) \neq U(L_i - 1) \end{cases} \tag{3.2}$$

After obtaining the related time parameters, the total execution time for a neural network $T(L, P, S(L \rightarrow P, TR)$ can be calculated via four different parameters, as shown in Eq. 3.3. Each layer L_i has a characteristic execution time on processor P_j, presented with a scheduling parameter $s(L_i)$. The transition cost $\tau(L_i, s(L_i), OUT|IN)$, is added to the total execution only if the result of layer L_i transitions to a new processor, represented by TR_i. Total energy consumption in Eq. (3.6) also has a closely similar pattern with the total execution time since the energy consumption is calculated by the same variables. We use TR_i boolean variable to indicate whether any transition occurs in Eq. (3.4) and to limit the number of transitions in Eq. (3.5).

$$\begin{split} T(L, P, S(L \rightarrow P), TR) = \sum_{i=0}^{len(L)} (t(L_i, s(L_i)) + \\ (TR_i \times \tau(L_i, s(L_i), OUT)) + (TR_i \times \tau(L_{i+1}, s(L_{i+1}), IN)) + \\ (TR_i \times pipeline(L_i, S(L_i)))) \end{split} \tag{3.3}$$

$$TR_i = \begin{cases} 1 & \text{if } S(i) \neq S(i+1) \\ 0 & \text{if } S(i) = S(i+1) \end{cases} \tag{3.4}$$

$$NumTransitions = \sum_{i=0}^{len(L)} TR_i \tag{3.5}$$

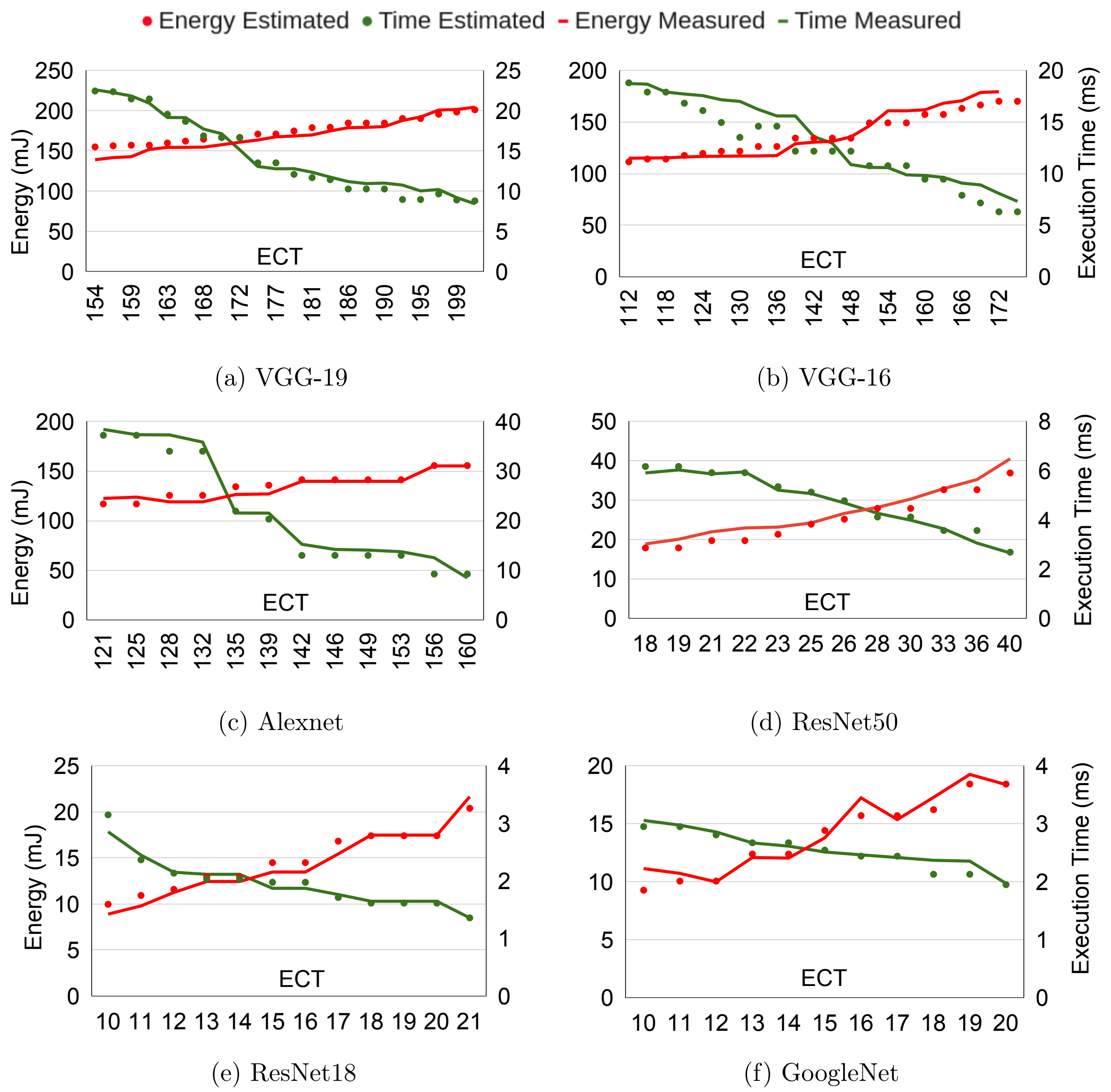
$$\begin{split} E(L, P, S(L \rightarrow P), TR) = \sum_{i=0}^{len(L)} e(L_i, s(L_i)) + \\ (TR_i \times e(L_i, s(L_i), OUT)) + (TR_i \times e(L_{i+1}, s(L_{i+1}), IN)) \end{split} \tag{3.6}$$

Our objective function is to minimize the execution time of a DNN inference for a given set of layers and processors, with each possible mapping of layers to the processors, and an energy constraint ECT. Thus, we define our objective function and the primary constraint as follows:

$$\begin{aligned} \min \quad & T(L, P, S(L \rightarrow P)) \\ \text{s.t.} \quad & E(L, P, S(L \rightarrow P)) < ECT \end{aligned} \tag{3.7}$$

3.6 Evaluation

The constraints described in Section 3.5 can be handled with an off-the-shelf constraint solver. We solve the objective function and its dependencies shown in Section 3.5 with the Z3 SMT solver, which efficiently determines satisfiability of logical/numeric constraints.



(a) VGG-19

(b) VGG-16

(c) Alexnet

(d) ResNet50

(e) ResNet18

(f) GoogleNet

Figure 3.5 The comparison between the energy and execution times of the schedules estimated by *AxoNN* versus the actual energy and execution times measured for the corresponding actual runs. For each network we have targeted varying ranges of ECT that is between the energy consumption of all-DLA and all-GPU execution.

3.6.1 Experimental Setup

In this study, we use Nvidia's Jetson Xavier AGX SoC since it embeds a performance-efficient (*i.e.*, GPU) and an energy-efficient (*i.e.*, DLA) DSA, together with access to the same shared DRAM memory. The software versions utilized on our experimental platform are Ubuntu OS 18.04, Cuda 10.2, TensorRT 7.1.3, CuDNN 8.0.0, ONNX 1.6.0, and TensorFlow 2.3.1. We use the TensorRT engine to optimize the pre-trained models collected from several neural network models, VGG-16/19 [160], Resnet18/50 [163], Alexnet [164], and GoogleNet[165]. The reason we focus on these networks is that all layers in these networks can be scheduled on both the GPU and the DLA. This allows us to flexibly explore all possible layer-to-DSA assignments, without TensorRT engine falling back to GPUs.

3.6.2 Experimental Results

We design an experiment to observe the effect of transitions after different types of layers on a DNN in Eq. 3.2. Since we mainly utilize two types of DSAs in our experiments (GPU and DLA), we first assign all layers to GPU, and measure the total execution time. Then, we repeat the experiment by applying a transition from GPU to DLA after the each layer on the neural network architecture and measure time on the DSAs. In order to analyze the effect of each transition, we subtract the time for each transition point from the previous transition experiment, and plot the results in Figure 3.6. In other words, for each L_i assigned to a different device, we calculate the execution time difference according to Equation 3.3.

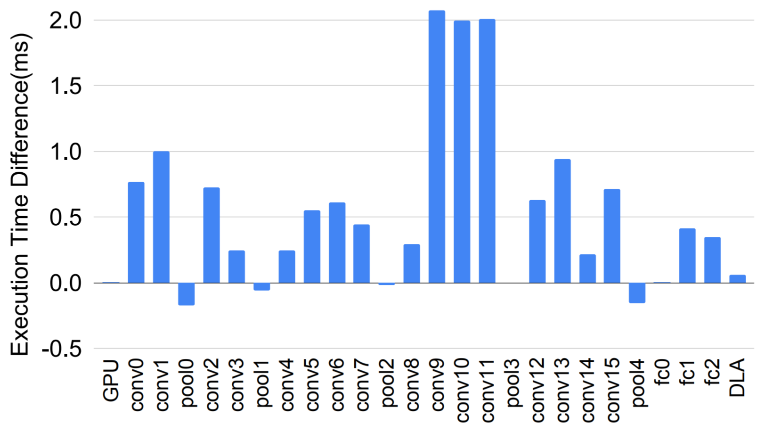


Figure 3.6 A transition is performed on the layer at the X-axis from GPU to DLA. As the number of layers increases on DLA, the execution time increases. Because the pipeline is broken, the transitions after pooling layers have negative values, as explained in Section 3.2.

Since the layer is running on a slower device (DLA), the difference in terms of the time is generally higher than zero in Figure 3.6. The layer-wise execution time characterization given in Figure 3.4 is inline with these results. The transitions after pooling layers have negative values. Since the effect of braking pipelines exceeds the slowdown of introduced by executing the next layer on DLA, the total execution time decreases. For this reason, applying a transition after such layers can provide an improvement in execution time.

We evaluate the feasibility of our model given in Section 3.5 with 6 different DNN models. We define our objective functions and the main constraint on the Z3 solver as given in Equation 3.7. The main idea here is to restrict the energy with an upper limit by minimizing the execution time of DNN inference. We provide the profiling results of layer execution time, energy, and transition characterization as inputs to the solver, and obtain the near-optimal schedule as output. Our results are given in Figure 3.5. X-axis shows the ECT constraint within a range from the minimum to the maximum amount of energy spent by all-DLA and all-GPU executions, respectively. On the left and right vertical axes, we represent energy consumption and execution times, respectively, corresponding to each value of ECT. The values represented by green dots correspond to the execution time estimates, whereas the green lines show the measured execution times when the schedule found by the solver for the specific ECT is executed. Similarly, the values represented by red dots and lines are for energy consumption. Overall, our results show that our model provides up to 97.1% accuracy on execution time prediction, and up to 98.2% accuracy on energy consumption prediction. In the worst cases, the execution time and energy prediction accuracy falls down to 78.1% and 71.9%, respectively.

3.6.3 Multi-transition and Scheduling Overhead

While each transition between DSAs costs extra time in the schedule, the most feasible execution may still include multiple transitions, *i.e.*, going back and forth between DSAs. Therefore, we run our solver by allowing more than a single transition, by increasing the value of the *NumTransitions* variable (Eq. 3.5) to 3. Table 3.2 lists the number of inter-DSA transitions that the near-optimal schedules include. The overhead of scheduling (*i.e.*, solver execution time) is under 5 seconds when *NumTransitions* is set to 1, and under 1 minute when *NumTransitions* is set to 3.

Table 3.2 The number of inter-DSA transitions that near-optimal schedules include when *NumTransitions* variable is set to 3.

Model	**1 Transition**	**2 Transitions**	**3 Transitions**
GoogleNet	19	3	0
VGG-19	16	4	2
VGG-16	18	4	0
Alexnet	10	2	0
ResNet50	8	4	0
ResNet18	9	3	0

3.7 Conclusion

This study presents *AxoNN*, a multi-accelerator execution scheme for heterogeneous SoCs. We explore the factors affecting the energy-aware scheduling of DNN workloads onto DSAs. We analyze the transition costs between DSAs in a shared-memory system and characterize the execution time and energy consumption of different DNN workloads. We build a scheduling model to find the minimum execution time for different energy targets. We test our methodology with 6 different networks, and test our results with the Z3 SMT Solver, obtaining up to 98% prediction accuracy.

CHAPTER 4
HAX-CONN: SHARED MEMORY-CONTENTION-AWARE CONCURRENT DNN EXECUTION FOR DIVERSELY HETEROGENEOUS SOCS

Two distinguishing features of state-of-the-art mobile and autonomous systems are: 1) There are often multiple workloads, mainly deep neural network (DNN) inference, running concurrently and continuously. 2) They operate on shared memory SoCs that embed heterogeneous accelerators tailored for specific operations. In this chapter, we will propose HaX-CoNN, a novel scheme that characterizes and maps layers in concurrently executing DNN inference workloads to a diverse set of accelerators within an SoC.

4.1 Introduction

Modern mobile and autonomous systems —such as cars, drones, and robots— hinge on edge intelligence, which involves running computationally demanding workloads [26, 166, 167]. Notably, a diverse range of applications embed multiple DNNs as subtasks such as object detection and semantic segmentation for autonomous systems [168, 169] or pose estimation and eye-tracking for VR applications [170, 171]. Workloads running *concurrently* and *continuously* in such systems necessitate powerful SoCs that can meet high computational demand and ensure safety [172] and QoS [173] requirements. Thus, such SoCs are often equipped with a CPU, a GPU, and one or more domain-specific accelerators (DSAs), optimized to perform specific types of operations. For example, NVIDIA Xavier AGX and Orin architecture comprise deep learning accelerators (DLA), and a programmable vision accelerator (PVA) in addition to CPU and GPU. Utilizing different types of DSAs for concurrently running workloads enables the flexibility to explore execution strategies [13]. *Leveraging this opportunity, in this work, we focus on mapping layers of parallel DNNs to different types of DSAs so that we can improve computational latency and system throughput.*

A common feature of such heterogeneous SoCs is the *shared physical main memory* where data is stored for access by all processing units (PUs) in the system. Even though this cost-driven design decision curbs costly data movement, memory subsystems in such architectures are often designed to accommodate the memory demands of a single PU at a time. Consequently, *shared memory contention emerges as one of the primary performance bottlenecks in mobile and autonomous SoCs, when parallel tasks are concurrently mapped to different accelerators* [16, 94, 174].

We propose *HaX-CoNN*, a *multi-accelerator and contention-aware execution scheme* for collaboratively and concurrently running DNNs on shared memory SoCs. *HaX-CoNN* is centered around characterizing common layers in DNNs according to their DSA-specific performance and identifying how they are affected by shared memory contention. Leveraging *decoupled* performance and contention characterization *at a* *layer-level*, *HaX-CoNN* exploits distinct capabilities of each DSA in the system by deciding whether the execution of the next layer in the DNN should *transition* to another DSA or not. *HaX-CoNN* uniquely finds an *optimal* mapping between the layers and DSAs in the system by formulating the problem as a set of constraint-based linear equations and utilizing SAT solvers to find a solution.

4.1.1 Contributions

Our work makes the following contributions:

- We present *HaX-CoNN*, a contention-aware, multi-accelerator execution scheme that *maximizes compute utilization* and *minimizes the overall latency* of concurrently running DNNs on shared memory SoCs.
- We propose a generalized and formal layer-to-accelerator mapping approach for concurrently running DNNs. We demonstrate that SAT solvers can be utilized to produce *optimal schedules* for *multi-accelerator* execution.
- We build a new contention modeling approach which significantly reduces profiling search space by decoupling performance measurement and the slowdown.
- We present D-*HaX-CoNN*, a dynamic runtime adaptation of SAT solver-based optimal schedule generation for dynamically changing workloads.
- We evaluate *HaX-CoNN* and D-*HaX-CoNN* on NVIDIA AGX Orin, Xavier AGX, and Qualcomm Snapdragon 865 SoCs. Our results show that *HaX-CoNN* can provide latency and throughput improvements up to 32% and 29%, respectively, over greedy-scheduling based approaches.

The remainder of this chapter is organized as follows: Section 4.2 surveys the background and key challenges of running multiple DNNs on shared-memory SoCs. Section 4.3 introduces the HaX-CoNN framework, covering layer grouping, per-layer characterization, contention modeling, and our optimal/dynamic scheduler. Section 4.4 outlines the experimental setup, including platforms, workloads, profiling, synchronization, and schedule generation. Section 4.5 presents a comprehensive evaluation across single- and multi-DNN scenarios, dynamic workloads, and exhaustive pairwise studies.

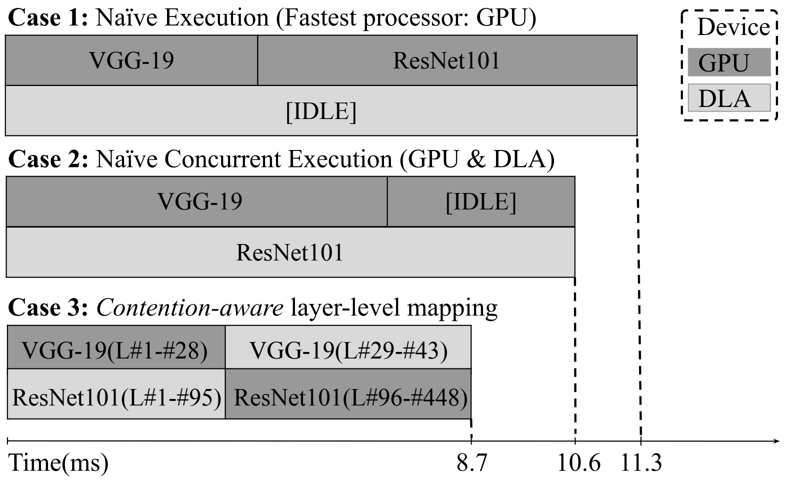


Figure 4.1 Different ways of executing VGG-19 and ResNet-101 DNNs in parallel on Xavier AGX.

4.2 Motivational Study and Related Work

We investigate concurrent execution on shared memory SoCs through a case study. In a typical loop running on autonomous systems, VGG-19 [175] and ResNet101 [163] can be used in tandem for vision (*i.e.*, perception) tasks. Since the remaining tasks in the autonomous loop depend on the completion of these two DNNs, utilizing all computational resources in the SoC for these DNNs is expected to reduce the total latency of the system. Figure 4.1 illustrates three different ways of executing two DNNs on NVIDIA Xavier AGX SoC. In *Case 1*, DNNs are *serially* executed on the fastest DSA, which is the GPU, resulting in 11.3ms of cumulative latency. However, this method leaves DLA idle, thereby under-utilizing the system resources. The first approach can be improved by a *naïve concurrent* execution, as shown in *Case 2*. In this scheme, VGG-19 is run on GPU, and ResNet101 is mapped to DLA, resulting in 10.6ms cumulative latency with a slight improvement. However, the speed-up obtained remains limited due to two reasons: (1) DLA takes longer to execute, leaving GPU idle towards the end, and (2) when GPU and DLA operate together, they contend for shared memory and slow down. *For more efficient execution of concurrent DNNs, we need a finer-grained, i.e., layer-level, mapping of the DNNs to DSAs.*

Case 3 depicts an ideal case where layers in both DNNs are divided into two groups after layers #28 and #95, respectively. For each DNN, the execution is switched between two DSAs at the boundary of corresponding layer groups (*i.e.*, transition point). While seemingly non-intuitive, this approach considerably improves the cumulative latency and increases the overall system utilization. This is due to a careful partitioning and mapping of layers to GPU and DLA in a way that: (1) the shared memory contention across concurrently running layers is minimized, (2) neither of the DSAs is left idle, and (3) the overhead of switching between accelerators between two layers is minimized. However, finding such partitioning is not trivial, and the state-of-the-art approaches (detailed in Section 2.1) fail to provide a holistic approach to perform this partitioning optimally.

Table 4.1 Feature comparison between the most related work and *HaX-CoNN*.

Features	Mensa [66]	AxoNN [38]	Pipeline [68]	OmniBoost [59]	MoCA [58]	Herald [56]	H2H [57]	***HaX-CoNN***
Concurrent DNNs	✗	✗	✗	✓	✓	✓	✓	✓
Multi-accelerator	✓	✓	✓	✗	✗	✓	✓	✓
Transition cost	✓	✓	✓	✓	✓	✗	✓	✓
Memory contention	✗	✗	✗	✗	✓	✗	✗	✓
Dynamic scheduling	✓	✗	✓	✗	✓	✗	✗	✓
Optimal schedules	✗	✓	✗	✓	✗	✗	✗	✓

Gap in the literature: Table 4.1 provides a snapshot of what the most relevant works offer and how they compare against *HaX-CoNN*. Achieving the ideal execution scenario with balanced layer distribution among the DSAs requires holistic consideration of several factors given in Table 4.1: (i, ii) interaction and mapping opportunities created by running *concurrent DNNs* on *different types of DSAs*, (iii) the *transition overhead* when the execution within a DNN switches across accelerators, (iv) the slowdown caused by the *shared memory contention* as layers run concurrently –our analysis shows that shared memory contention-unaware decisions can reduce system performance by up to 70%, as detailed in Section 4.5.2–, (v) support for dynamic schedules, and (vi) optimal schedule creation. The efficient and safe operation of performance critical mobile and autonomous workloads on shared memory SoCs depends on the *holistic* consideration of all these factors. Our experiments demonstrate that the lack of such consideration results in mispredicted performance, which in turn results in inefficient execution.

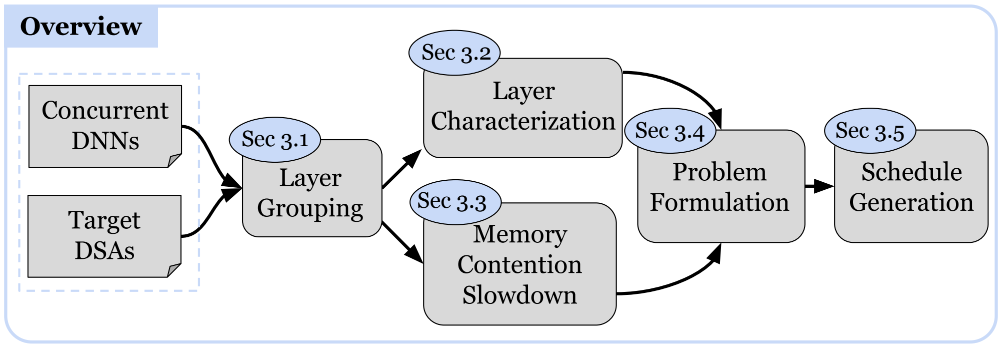


Figure 4.2 Overview of *HaX-CoNN*.

4.3 HaX-CoNN: Heterogeneity-aware Execution of Concurrent Deep Neural Networks

An overview of our proposed methodology is given in Figure 4.2. *HaX-CoNN* takes the *DNNs* to be scheduled and the target *DSAs* as input and produces the optimal schedule as output.

4.3.1 Layer Grouping

The first step involves identifying minimal layer groups to serve as atomic assignment units for DSAs. This grouping considers several factors. After we identify all feasible layer groupings, hence the *transition points*, the next step is to characterize each layer's (or layer group's) performance and the overhead if an inter-DSA transition occurs after that particular layer (or layer group).

1) Preserving layer optimizations Layer/operator fusion [176–178] merges multiple layers into a single layer. *Transition points* during DNN execution, where we switch execution from one DSA to another, should not impede operator fusion. Therefore, we ensure that fusible layers are grouped together and mapped to the same accelerator.

2) Input/output reformatting DSAs typically operate in an internal hardware (HW) pipeline. If a transition from a layer mapped to such a DSA disrupts its pipeline, then an additional output reformatting operation is inserted by the execution framework. Similarly, input reformatting might be required after transitioning to that DSA. Layer groupings can be structured to avoid such formatting overheads.

3) Accelerator and software limitations DSAs are often limited by the layer types, parameters, and batch sizes they support. DNN execution frameworks, such as NVIDIA TensorRT [161] and Qualcomm SNPE [179], ensure such constraints are followed. We identify such limitations via vendor specific API calls and our framework considers these limitations when locating valid transitions between accelerators.

In this step, we group layers as follows: If transitioning to another DSA after a layer is prohibited or leads to increased overhead, the layer is grouped with subsequent layers. Otherwise, the layer is marked as a potential *transition point.*

4.3.2 Per-layer Performance Characterization

Strategically assigning layers to the DSAs where they will run most efficiently has the potential to increase performance. Previous studies [56, 66, 180–182] have detailed various parameters that affect the efficiency of deep learning accelerators, such as layer type, input size, kernel size, etc. Different layers within a DNN yield varying performance speed-ups when run on a specific DSA. To illustrate and analyze this further, we conduct an experiment where we profile layer groups in GoogleNet on GPU and DLA. The results given in Table 4.2 show that while the DLA performs slower than GPU for all layers, the speed reduction is less severe for some layers. The fourth column lists the ratio of execution time on DLA over

GPU, which varies from 1.40x to 2.02x among different layer groups. Larger performance discrepancies primarily arise because compared to DLAs, GPUs are heavily optimized for large-size matrix operations and they are capable of more effectively exploiting performance on convolution operations with larger inputs. Conversely, smaller kernels, such as those in groups 95-109 and 124-140, are better fits for the DLA's internal on-chip buffer.

Table 4.2 Execution (E) and transition (T) time of layer groups in GoogleNet

Layer Group	GPU	DLA	D/G	Transition (G to D)	Transition (D to G)	Memory Thr. (%)
0-9	0.45	0.75	1.65	0.056	0.15	41.97
10-24	0.19	0.34	1.80	0.075	0.13	62.21
25-38	0.31	0.45	1.44	0.062	0.08	78.49
39-52	0.18	0.37	2.02	0.011	0.03	53.41
53-66	0.16	0.31	1.98	0.055	0.03	55.70
67-80	0.17	0.33	1.96	0.024	0.04	59.24
81-94	0.21	0.31	1.50	0.058	0.05	62.60
95-109	0.25	0.35	1.40	0.030	0.06	76.12
110-123	0.16	0.27	1.66	0.024	0.07	66.95
124-140	0.24	0.36	1.49	0.007	0.05	47.96

Prior studies [38, 59, 66, 183, 184] show that it is feasible to characterize DNNs via a *layer-centric* profiling approach where commonly used layer types are profiled beforehand for different input and filter sizes. Following a similar methodology, we profile each layer or layer group on the DSAs in the system. We utilize IProfiler interface of TensorRT on NVIDIA devices [185], which reports per layer time. Profiled execution times are then embedded into variable t, which is used in equations 4.2, 4.4, 4.5, and 4.7 in Section 4.3.5.

4.3.3 Inter-DSA layer Transitions Characterization

Despite the potential performance boost offered by multi-DSA execution, transitioning between DSAs comes with a cost. This cost, crucial for accurate performance predictions and optimal scheduling, is contingent on the size of the transient data in private caches of DSAs. The output of the layer preceding the transition is flushed back to the shared memory so that the DSA where the next layer will execute on can access it. The fifth and sixth columns in Table 4.2 represent the time spent when the execution, after each layer group, switches from GPU to DLA and vice versa. As output data sizes decrease toward the end of layer groups, so does transition time. Notably, our experiments also reveal that some layer groups, such as 39-53 and 95-109, ending with pooling layers result in significantly less transition overhead when switching from GPU to DLA. We empirically derive the transition costs of the layers on our target set of accelerators, following the methodology outlined in [38]. To implement them, we insert $MarkOutput$ and $addInput$

API calls in TensorRT [185]. We then incorporate them into equations 4.2 and 4.3 in Section 4.3.5.

4.3.4 Characterizing Shared Memory Contention

One of the core novelties of our work is its ability to account for the slowdown caused by shared memory contention. Since existing multi-DSA schedulers do not consider this when making scheduling decisions, the resulting mappings often leave the system under-utilized. However, estimating this slowdown, especially for multi-DSA systems, is not trivial. Exhaustive and peer-wise runs of all combinations of layers by colocating them are required. This will result in a factorial explosion of profiling search space and require significant profiling time [186].

To prevent this, we follow a *decoupled* two-step approach: we first characterize each layer's requested memory throughput when they are run standalone. Using these throughput values, we then utilize a processor-centric slowdown model, PCCS [174], to estimate the slowdown without relying on layer-specific information. PCCS represents the slowdown between multiple concurrent workloads as a function of requested memory throughput and external memory traffic, and builds a piece-wise model to predict the slowdown experienced by the accelerator requesting the throughput. Built upon PCCS, our decoupled approach is performed at layer-level and separates the collection of layer-specific standalone performance profiles, as collected in Section 4.3.2, and the slowdown caused by concurrent execution.

The last column in Table 4.2 lists memory throughput measurements per layer group in GoogleNet. The general pattern we observe in many DNNs is that higher input size results in higher memory throughput. We also observe that, as the filter size in convolution and pooling layers gets larger, there is a decrease in throughput due to the increasing arithmetic intensity of the underlying operation.

Conventional hardware counters to monitor requested memory throughput may not be applicable for some *black-box* DSAs which cannot be profiled with conventional tools. For example, NVIDIA Nsight Compute tool [187] can profile requested memory throughput on GPUs but not on DLAs. As an alternative way to methodologically solve this issue, we develop a four-step approach: 1) We first profile target layers on GPU and analyze the memory throughput for several layer types (*i.e.*, convolution, pooling, and fully connected) and their parameters (*i.e.*, input size filter size). Throughout their execution lifetimes, we observe that many layers individually exhibit homogeneous memory access characteristics as they internally embed homogeneous and dense computations. 2) We then profile external memory controller (EMC) utilization for all layers on both DLA and GPU. In Figure 4.3, the input sizes of i1-i5 for the convolution layers correspond to (224,224,64), (224,112,64), (112,112,64), (112,56,64), (56,56,64) and filter sizes of f1-f5 correspond to (1x1), (2x2), (3x3), (4x4), (5x5), respectively. Our analysis reveals that the EMC utilization for DLA and GPU are correlated and proportional. 3) Using this observation, we estimate its memory throughput on black-box DSAs (*e.g.*,, DLA in this case) by dividing its GPU-based memory throughput by the ratio of EMC utilization of GPU and DSA for that specific layer. 4) Finally, by utilizing PCCS, we estimate the slowdown of a layer on an accelerator via its requested memory throughput and the external memory throughput requested by the other concurrently running layer on the other DSA.

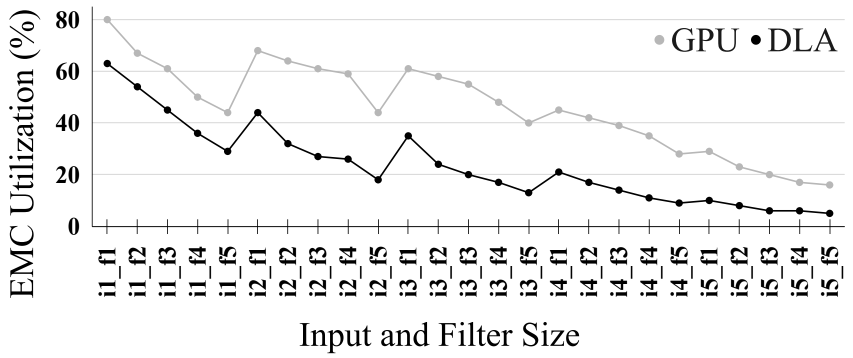


Figure 4.3 EMC utilization by conv layers on GPU and DLA with varying input (i) and filter (f) sizes

When multiple layers are run simultaneously on different accelerators, the degree of slowdown throughout their execution is non-uniform and depends on the other layers running concurrently. Figure 4.4 illustrates this behavior by depicting the execution timelines of five hypothetical layers belonging to three different DNNs. Each timeline represents the execution of ith layer on jth DNN, labeled as L_{ij} on DSA_k. The black regions in the timeline represent the time that the executions would take if all layers were run separately. Each colored extension to the black regions indicates slowdowns for different sets of layers running together. To address the complexity of handling varying amounts of slowdown during the execution of collocated layers, we introduce a scheduling concept called *contention interval.* Each contention interval (t_i, t_{i+1}) represents a period separated by the start or end of a layer execution, and it is represented by the Eq. 4.8 explained in Section 4.3.5. During each contention interval, different rates of slowdowns are observed by each layer, and the slowdown depends on the cumulative external memory pressure demands during that interval.

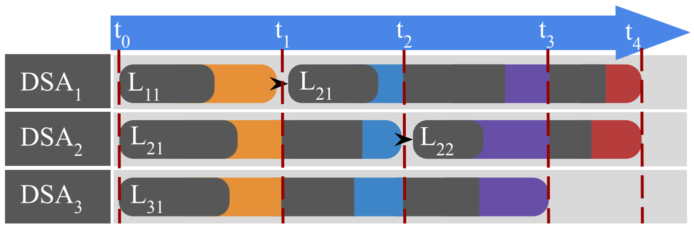


Figure 4.4 Illustration for a hypothetical execution of five layers from three DNNs running on three different accelerators. Colored regions indicate additional slowdowns each layer experiences for varying external memory pressure.

4.3.5 Formulating the Problem

We integrate layer execution time, inter-accelerator transition time, and memory contention slowdown into a cost function and formulate the scheduling problem as a series of linear equations. Table Table 4.3 summarizes the variables and notations we use in the formulation. The primary input to our model is the DNN set for which we explore its mapping to the accelerator set A. $L_{i,n}$ denotes the smallest layer entity that belongs to the layer set of DNN_n. A layer entity is either a single layer or a group of layers, as explained in Section 4.3.1. Functions $t(L_{i,n}, a)$ and $\tau(L_{i,n}, a, OUT|IN)$ represent the execution time and transition overheads of layer $L_{i,n}$ on accelerator A_a, respectively.

Table 4.3 The notation used by our formulation.

Notation	**Explanation**
DNN_n	nth DNN in the given DNN set which contains networks to be executed concurrently
$L_{i,n}$	ith layer of the nth DNN in the DNN set
$len(DNN_n)$	Total number (length) of layer groups in DNN_n
A_a	ath accelerator in the given accelerator set A
$\mathrm{S}(L_{i,n})$	The schedule, *i.e.*, accelerator mapping, of $L_{i,n}$
$t(L_{i,n}, A_a)$	Total execution time of $L_{i,n}$ on A_a
$st(i,n)$	Execution start time of $L_{i,n}$
$et(i,n)$	Execution end time of $L_{i,n}$
$\tau(L_{i,n}, A_a, OUT\|IN)$	The time required to transition the DNN execution after|before layer l_i executed on accelerator A_a
$TR_{i,n}$	Boolean var if a transition is set after layer $L_{i,n}$
$T(L, S(L))_n$	Total execution time elapsed by the execution of given sets of layer L of the nth DNN
$c_{L_{i,n},S(L),L}$	The slowdown of $L_{i,n}$ due to the contention caused layers running on other accelerators, *i.e.*, $S(L)$
$I_{i,j}$	The length of interval where layers i and j overlap
Int	Interval array holding start and end time of layers

The goal of our formulation is to find the schedule S for all layers across all DNNs. The schedule function, defined in Eq. 4.1, returns A_a that $L_{i,n}$ should be mapped to. S is assumed to be initially unknown and will be determined by the solver later.

$$\begin{gathered} S(L_{i,n}) = A_a \ \ where \ \ 1 \le i \le len(DNN_n), \\ 1 \le n \le len(DNN) \ \ , \ \ 1 \le a \le len(A) \end{gathered} \tag{4.1}$$

Total execution time of a DNN is formulated in Eq. 4.2. Total time comprises *standalone execution time* t of each layer, the *slowdown* C, and and IN and OUT *transition costs*, τ.

$$\begin{gathered} T(L, S(L \rightarrow A))_n = \sum_{i=0}^{len(DNN_n)} t(L_{i,n}, S(L_{i,n})) * C_{L_{i,n},S(L),L} \\ + \ TR_{i,n} \ \times \ \tau(L_i, s(L_i), OUT) + \ \ TR_{i,n} \ \times \ \tau(L_{i+1}, s(L_{i+1}), IN) \end{gathered} \tag{4.2}$$

We encode the decision to make transitions into our formulation via the boolean function given in Eq. 4.3. This function compares the accelerator assignments of adjacent layers $L_{i,n}$ and $L_{i+1,n}$. If the assignments differ, the transition cost, τ, is subsequently incorporated into Eq. 4.2.

$$TR_{i,n} = \begin{cases} 1 & \text{, if} \quad S(L_{i,n}) \neq S(L_{i+1,n}) \\ 0 & \text{, if} \quad S(L_{i,n}) = S(L_{i+1,n}) \end{cases} \tag{4.3}$$

Eq. 4.4 and 4.5 compute the execution start and end times, $st()$ and $et()$ respectively, for layer $L_{i,n}$. Int array in Eq. 4.6 stores the start and end time for layers, facilitating the iterative comparison of the contention intervals across layers.

$$st(i,n) = T(L_{0 \ to \ i-1 \ , \ n}, S(L))_n \tag{4.4}$$

$$et(i,n) = st(i,n) + t(L_{i,n}, S(L_{i,n})) * C_{i,n} \tag{4.5}$$

$$\begin{gathered} \forall L_{i,n} \ \ , \ \ [st(i,n), et(i,n)] \in Int \\ where \ \ 1 \le i \le len(DNN_n), \ \ 1 \le n \le len(DNN) \end{gathered} \tag{4.6}$$

The contention function C, outlined in Eq. 4.7, calculates the total slowdown for layer $L_{i,n}$ by taking each time overlapping with that layer and the slowdown ratio corresponding to the interval. The contention model returns an estimated slowdown amount depending on the bandwidth demanded by layer l_i and cumulative external bandwidth demanded by other layers running inside the same interval.

$$\begin{gathered} C_{L_{i,n},S(L),L} = \sum_{I_k \in Int} \frac{I(L_{i,n} \ , \ L_{j,n}) * cont\_model(L_{i,n} \ , \ L_s)}{t(L_{i,n}, S(L_{i,n})) * len(L_s)} \\ where \ 1 \le j \le len(DNN_n), \ 1 \le n \le len(DNN), \ L_{j,n} \ \in \ L_s \\ Int_k \ \cap \ [st_{i,n}, et_{i,n}] \ \neq \ \emptyset \ , \ Int_k \cap [st_{j,n}, et_{j,n}] \neq \ \emptyset \end{gathered} \tag{4.7}$$

Eq. 4.8 details how we determine the duration of contention intervals. If a layer faces no contention, the equation simply returns the layer's execution time, leading to a value of 1 to be returned in Eq. 4.7, thereby indicating no slowdown effect for a layer running independently in Eq. 4.2.

$$I(i,j) = \begin{cases} e_j - s_i & if(s_j \leq s_i \leq e_j \quad \& \quad s_i \leq s_j \leq e_i) \\ e_j - s_j & if(s_i \leq s_j \leq e_i \quad \& \quad s_i \leq s_j \leq e_i) \\ e_i - s_j & if(s_i \leq s_j \leq e_i \quad \& \quad s_j \leq e_i \leq e_j) \\ e_i - s_i & if(s_i \leq s_j \quad \& \quad e_i \leq e_j) \\ e_i - s_i & otherwise \end{cases} \tag{4.8}$$

We establish a constraint in Eq. 4.9 that limits two distinct layers from sharing the same accelerator for longer than an ε interval. Ideally, in a flawless model, the estimated execution and slowdown values could yield perfect transitions where accelerator usage periods can be precisely predicted. Variable ε allows us to mitigate the prediction errors, and facilitates more transition points by allowing for a tiny overlap of concurrently assigned layers on the same accelerator at the start or end of their executions.

$$\begin{gathered} L_{i,nn}, L_{j,n} \ (L_{i,nn} \in DNN_{nn} \ \ and \ \ L_{j,n} \in DNN_n \ \mid \\ st_{L_{j,n}} < st_{L_{i,nn}} \mp \varepsilon < et_{L_{i,nn}} \ or \ st_{L_{j,n}} < et_{L_{i,nn}} \mp \varepsilon < et_{L_{j,n}}) \\ where \ S(L_{i,nn}) = S(L_{j,n}) \ \ , \ \ nn \neq n \end{gathered} \tag{4.9}$$

Objective functions: Depending on the different scenarios that a user may target, we propose two separate objective functions: Equation 4.10 maximizes the utilization of the system to increase the total throughput and Equation 4.11 minimizes the maximum latency among DNNs. The use cases for objective functions are further elaborated in Section 4.5.

$$\max \sum_{n=1}^{len(DNN)} \frac{1}{T(L, S(L))_n} \tag{4.10}$$

$$\min \max \ T(L, S(L))_n \tag{4.11}$$

4.3.6 Optimal and Dynamic Schedule generation

In our work, we target optimal schedules that satisfy given objectives and constraints because we don't resort to heuristics to find such schedules. We achieve this by representing the entire scheduling problem formulated in Section 4.3.5 as a constraint-based optimization problem and solving with industry-strength SAT solvers such as Z3 [188], Gurobi [189], and OptiMathSAT [190]. These solvers employ branch & bound techniques to converge towards optimal solutions for many NP-complete problems (*i.e.*, job-shop scheduling) [191]. Considering the relatively small parameter search space of our targeted problem set (*i.e.*, total number of accelerators and tasks in the system), the use of SMT solvers provides optimal schedules in seconds. Depending on the operational requirements of the autonomous system, optimal schedules can be found either statically or dynamically.

Generating optimal schedules beforehand (*i.e.*, *statically*) is feasible for a variety of scenarios, such as in autonomous systems with fixed resolution input devices (like cameras and lidars) and many DNNs designed for a fixed image or a video frame size. Some scenarios, such as a drone switching between *discovery* or *tracking* modes, might require unique control flow graphs (CFGs). Such CFGs (or the path followed in a CFG) and their corresponding schedules can be pre-determined statically and toggled during the execution. Thus, users of *HaX-CoNN* can rely on offline profiling to determine the execution costs needed for static scheduling [192].

There are other cases where the static generation of optimal schedules may not be possible. For example, different DNN models may be required for various phases of the autonomous system execution [193–195], resulting in an unpredictable change in the CFG. For such scenarios, we propose D-*HaX-CoNN*, a runtime-based adaptation of our solution to (1) run SAT solvers on-the-fly, (2) gradually achieve and apply better schedules, and (3) eventually reach an optimal solution as the autonomous system continues to operate. This approach is feasible because autonomous systems often embed long-running loops and once an optimal schedule is found for a recently changed CFG, it will be reused for a while.

4.4 Experimental Setup

In this subsection, we list the several computing platforms and DNN architectures we have evaluated on HaX-CoNN. We also detail how we performed profiling, synchronized multiple DNN execution and generated the schedules through solver.

Table 4.4 The HW specifications for NVIDIA AGX Orin

NVIDIA AGX Orin	
GPU	Ampere arch. 1792 CUDA & 64 Tensor cores
DSA	NVDLA v2.0
CPU	12-core Arm Cortex v8.2 64-bit
Memory	32GB LPDDR5 | **Bandwidth: 204.8 GB/s** with 256-bit
Software	JetPack 5.0.1

Table 4.5 The HW specifications for NVIDIA Xavier AGX

NVIDIA Xavier AGX	
GPU	Volta arch. 512 CUDA and 64 Tensor cores
DSA	NVDLA v1.0
CPU	8-core Carmel Arm v8.2 64-bit
Memory	16GB LPDDR4 | **Bandwidth: 136.5 GB/s**, 256-bit
Software	JetPack 4.5

Table 4.6 The HW specifications for Qualcomm 865 Mobile Development Kit

Qualcomm 865 Mobile Development Kit	
GPU	Qualcomm Adreno™ 650 GPU
DSA	Hexagon 698 DSP
CPU	Qualcomm Kryo 585, 8-core, up to 2.84GHz
Memory	6GB LPDDR5 | **Bandwidth: 34.1 GB/s** with 64 bits

4.4.1 Computing Platforms

We use three popular heterogeneous SoCs to evaluate *HaX-CoNN*: NVIDIA AGX Orin [196], Xavier AGX [197], and Qualcomm SnapDragon 865 development kit [198]. All three platforms have a shared memory with multiple accelerators. The technical specifications of these systems are summarized in Table 4.4, Table 4.5, and Table 4.6. It is essential to note that the maximum number of accelerators we consider in our experiments is limited to two because, to the best of our knowledge, there are no off-the-shelf SoCs that offer more than two types of programmable DSAs for DNN acceleration.

4.4.2 Applications

We use the DNNs that are commonly used in benchmarking DNN inference: Alexnet [199], GoogleNet[200], Inception-V4 [201], ResNet18/52/101/152 [163], VGG-19[175], FCN-ResNet18, CaffeNet [202], DenseNet [203], and Inc-Res-v2 [204] with datasets from COCO [205], ImageNet ILSVRC [206], and Cityshape [207]. These DNNs could be used for various tasks in autonomous systems, such as object detection, image recognition, semantic segmentation, pose estimation, and depth estimation [168]. Standalone runtime performances are given at Table 4.7.

Table 4.7 Standalone runtimes (ms) and relative performance.

DNN & Device	**NVIDIA AGX Orin**		**NVIDIA Xavier AGX**	
	GPU (ms)	DLA (ms)	GPU (ms)	DLA (ms)
CaffeNet	0.74	1.79	2.26	5.51
DenseNet	2.19	3.10	7.84	-
GoogleNet	0.99	1.52	1.98	3.68
Inc-res-v2	3.06	5.15	15.12	17.95
Inception	2.49	5.66	8.31	15.94
ResNet18	0.41	0.74	1.37	2.81
ResNet50	0.91	1.67	2.88	6.01
ResNet101	1.56	2.47	5.34	10.6
ResNet152	2.19	3.26	7.7	12.71
VGG19	1.07	2.93	5.95	19.05

4.4.3 Profiling

Profiling duration varies by platforms: Computation, transition, and contention characterizations can take up to 3, 10, and 15 minutes per DNN model, respectively, on NVIDIA Orin, Xavier, and Qualcomm platforms whereas building engines require more time on NVIDIA boards. Since our approach is layer-centric, we performed profiling only once and it is offline.

Neural network synchronization: TensorRT natively does not provide support synchronization between the layers of DNNs concurrently running at different DSAs. To make sure that the inter-accelerator transitions across DNNs are properly performed, we implement a TensorRT plugin that employs inter-DNN synchronization via inter-process shared memory primitives.

4.4.4 Schedule Generation

We solve our formulation given in Section 4.3.5 by using Z3 SMT solver. Z3 has shown superior performance for scheduling problems over popular solvers [191]. It works by determining the satisfiability of the constraints and finding an optimal solution for a given objective and constraints. In most of our experiments, Z3 takes under three seconds to run on a single CPU core of NVIDIA Orin AGX. In some cases, such as for the Inception-ResNet-v2 network which consists of 985 layers, the solver takes around ten seconds to find the optimal schedule.

4.5 Evaluation

We demonstrate the utility of *HaX-CoNN* via four execution scenarios with different objectives and also via an experiment that exhaustively collocates all the DNNs in our evaluation set. Scenario 1 aims to maximize throughput in concurrent data processing on the same DNN whereas scenarios 2 and 3 target two different DNNs operating in parallel and in a pipeline fashion, respectively. Scenario 4 is a hybrid of scenarios 2 and 3. We benchmark *HaX-CoNN* against five different baselines: (1) GPU only, (2) non-collaborative GPU & DLA, (3) Mensa [66] (which only supports single-DNN execution), (4) Herald [56], and (5) H2H [57] (which both support multi-DNN execution).

4.5.1 Running Multiple Instances of the Same DNN

Edge deployments frequently spin up several replicas of a single DNN to drive throughput or segregate service-level priorities. However, running such an operation intensifies both accelerator imbalance and shared-memory contention.

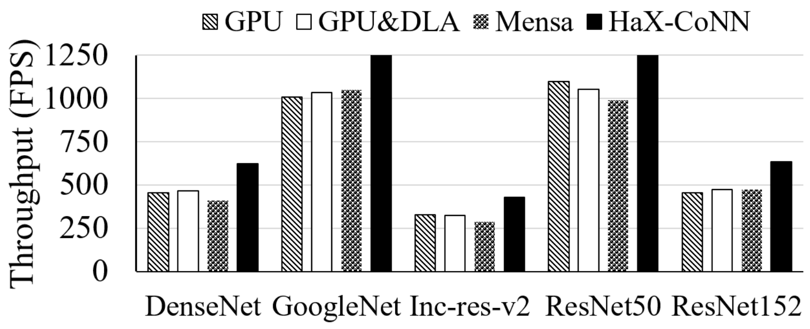


Figure 4.5 Throughput (FPS) comparison for Scenario 1: Multiple instances of the same DNN is run concurrently on NVIDIA AGX Orin.

Scenario 1 - Concurrent image processing with same DNNs: In systems aiming for high throughput, multiple instances of the same DNN could concurrently process consecutive images. Figure 4.5 reports the results of five different experiments designed for this scenario. The experiments are run on NVIDIA Orin and we compare *HaX-CoNN* against two naïve baselines and Mensa [66]. Overall, our experiments for this scenario show that *HaX-CoNN* can boost throughput (*i.e.*, *FPS*) up to 29%. There are several key observations we make in this experiment: (1) In GoogleNet experiment, *HaX-CoNN* maps the middle groups of layers (1-95 and 38-149) to GPU for both DNN instances since GPU executes those layers ∼2x faster than the DLA. (2) Due to shared memory contention, non-collaborative GPU & DLA execution does not always generate a better throughput compared to GPU-only execution. (3) We observe either limited improvements or no improvement by Mensa as it doesn't consider shared memory contention, leading to mismatched layer transitions. Even though Mensa considers transition costs, its greedy strategy fails to account for the transition costs occurring in the future, leading to inaccurate transition decisions.

4.5.2 Concurrently Running Different Type of DNNs

Table 4.8 lists the results of the experiments we performed by comparing *HaX-CoNN* to naïve and state-of-the-art multi-DNN concurrent execution schemes. Experiments 1-5 are on Xavier AGX, 6-8 are on AGX Orin, and 9-10 are on Qualcomm 865. The second to fourth columns describe the experiment designs and the corresponding scenarios. There are four baselines we compare our work against: (1) *GPU-only*, (2) *GPU & DSA*, (3) *Herald* [56], (4) *H2H* [57]. The last three columns list the optimal schedules found by *HaX-CoNN*, the latency and throughput (*i.e.*, FPS) for *HaX-CoNN*, and the improvement over the best-performing baseline.

Table 4.8 Experiments run for Scenarios 2, 3, and 4. We compare these scenarios against baselines when run on NVIDIA Xavier AGX (in experiments 1-5), NVIDIA AGX Orin (in experiments 6-8), and Qualcomm 865 (in experiments 9-10). DSA refers to DLA for NVIDIA platforms and to the Hexagon DSP for the Qualcomm platform.

Exp #	Goal	DNN-1	DNN-2	(1) GPU only		(2) GPU & DSA		(3) Herald		(4) H2H		Optimal schedule by HaX-CoNN		Runtime of MC^3 schedule	
				Lat.	FPS	Lat.	FPS	Lat.	FPS	Lat.	FPS	TR	Dir.	Lat.	FPS
1	Min Lat.	VGG-19	ResNet152	17.05	58	**16.05**	62	19.73	50	16.55	60	29 89	DtoG GtoD	**13.01**	77
2	Min Lat.	ResNet152	Inception	16.23	61	15.96	62	15.81	63	**15.75**	64	188 72	DtoG GtoD	**13.11**	76
3	Max FPS	Alexnet	ResNet101	11.04	90	10.97	**93**	12.10	82	11.49	87	11 161	GtoD DtoG	8.7	**115**
4	Max FPS	ResNet101	GoogleNet	7.02	**143**	7.37	140	8.95	111	9.10	109	0 0	DtoG DtoG	7.02	**143**
5	Min Lat.	GoogleNet ResNet152	FC_Res18	**15.41**	77	18.88	61	23.68	47	20.90	54	38 235	DtoG GtoD	**12.09**	85
6	Min Lat.	VGG-19	ResNet152	**3.95**	267	4.58	218	5.76	174	4.90	204	27 95	DtoG GtoD	**3.21**	311
7	Max FPS	GoogleNet	ResNet101	4.12	378	4.24	364	4.44	340	4.13	**380**	38 128	DtoG GtoD	3.4	**426**
8	Min Lat.	ResNet101 GoogleNet	Inception	5.06	197	4.97	201	5.56	180	**4.91**	203	31 88	DtoG GtoD	**4.41**	226
9	Max FPS	GoogleNet	ResNet101	98.3	10.1	79.1	**12.6**	95.9	10.4	113.8	8.8	52 148	DtoG GtoD	71.08	**14.1**
10	Min Lat.	Inception	ResNet152	219.6	4.5	**178.2**	5.6	223.1	4.5	202.3	5.2	17 135	DtoG GtoD	**155.3**	6.4

Scenario 2 - Two different DNNs operating on the same data: This scenario illustrates a case where different DNNs, such as object detection and image segmentation, process the same input in parallel, and they synchronize afterwards. The results are assumed to be passed on to subsequent tasks, such as motion planning [208], and then the loop is started over. Experiments 1, 2, 6, and 10 in Table 4.8 are run to demonstrate this scenario on three different target architectures. Our results show that *HaX-CoNN* improves both latency and throughput up to 23% in all four experiments of this scenario. We also observe that both H2H and Herald make inaccurate latency estimations that are wrong by up to 75% since neither of them considers shared memory contention. Experiments 1 and 6 show that *HaX-CoNN* results in different schedules for the same scenario running on different SoCs. For experiment 10, GPU & DSP is the best performing baseline for Qualcomm platform since GPU & DSP are more balanced on this platform in terms of their computation capability. Even though the schedule found by *HaX-CoNN* in experiment 10 on Qualcomm has a relatively higher transition cost among other transition candidates, the improvement primarily comes from minimizing the memory contention and effectively distributing the layers to DSAs.

Scenario 3 - Two different DNNs operating on streaming data: This scenario examines a common autonomous system setup where the input (*e.g.*,, camera stream) is available as a data stream and multiple tasks, such as object detection followed by object tracking [209], are executed in a pipelined manner. This scenario is covered by experiments 3, 4, 7, and 9. To establish the dependency among DNNs, we connect the last layer of the DNN_1 to the first layer of DNN_2 as an input. Interestingly, *HaX-CoNN* opts not to use DLA for none of the layers in experiment 4 since running two images sequentially on the GPU yields a higher throughput. Particularly, the performance of DLA on ResNet18 is less than the slowdown imposed on the GPU. Overall, if there are cases where layer-level mapping does not foster any benefits, *HaX-CoNN* is capable of identifying these cases and utilizing the baseline solution instead. Our scheme guarantees that no worse results are obtained than the naïve baselines.

Scenario 4 - Multiple DNNs with concurrent and streaming data: In this scenario, two DNNs (DNN_1 and DNN_2) have a serial dependency in between and another DNN (DNN_3) runs in parallel with the former two [209]. Experiments 5 and 8 belong to this scenario and the objective function is set to minimize the combined latency. *HaX-CoNN* is able to provide latency and throughput improvements up to 22%. Best performing baselines run DNN_3 mostly on GPU since unbalanced workloads among accelerators and shared memory contention alleviate the advantages of concurrent utilization. In experiment 5, the schedules that use both DSAs concurrently perform worse than serialized GPU executions, since DLA is generally less effective in running fully-connected layers. In experiment 8, H2H provides the fastest baseline performance since they are capable of exploiting heterogeneity of DSAs for appropriate layers (*e.g.*,, such as running DLA-efficient small layers on DLA and assigning others to the GPU). However, the schedules proposed by H2H lead to an over-subscribed DLA execution. On the other hand, *HaX-CoNN* finds the right transition points where no accelerators are overloaded and the transition cost is lower. In some schedules generated by H2H, such as experiment 3, while the transition points that are identified by H2H prevent accelerator over-subscription, the assignments for the remaining layers on both DNNs lead to workload imbalance.

Throughout the experiments, we generally observe that the benefits of architectural heterogeneity exploited by the state-of-the-art remain limited. The primary reason for the subpar performance of H2H and Herald compared to naive baselines is that their cost functions ignore shared memory contention. This, in turn, causes the timings to be mispredicted and eventually results in being unable to generate optimal schedules. Certain layers end up being assigned to the same accelerator (either GPU or DSA) at the same time, and this is due to poor (*i.e.*, non-optimal) handling of constraints triggered by mispredicted execution times. For example, on the DLA, two layer groups that are supposed to execute at different times are scheduled together, but they end up waiting for each other. During this time, the other accelerator (*e.g.*,, the GPU) is left idle.

In experiments 4 and 9 of Table 4.8, the latency of schedules generated by Herald is better than H2H because H2H makes optimizations to reduce the transition costs, yet leading to worse inter-DSA contention. We also observe that some of the optimizations performed by H2H are already performed by TensorRT. Therefore, such optimizations are already covered by our baselines, and this may hinder the benefits of H2H over our baselines. On the other hand, this situation does not affect the benefits demonstrated by *HaX-CoNN* over H2H and Herald. Also, it is worth noting it takes more time to generate schedules with H2H or Herald (*i.e.*, more than 10 seconds in most cases) than with HaX-CoNN.

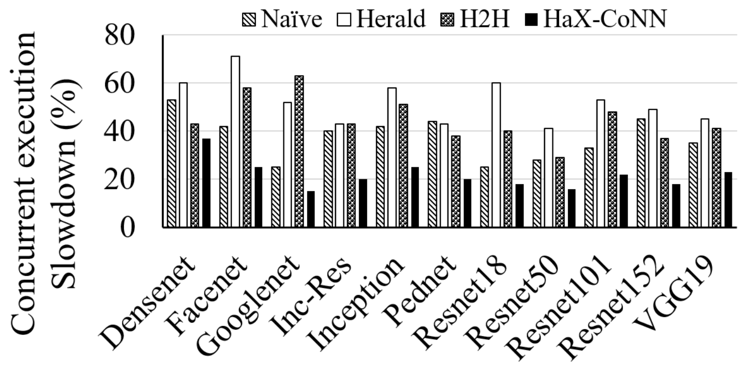


Figure 4.6 Slowdown of concurrently executing GoogleNet on GPU with different DNNs on DLA.

Based on what we observe in Table 4.8, we further analyze the slowdown caused by memory contention. Figure 4.6 depicts the amount of slowdown experienced by GoogleNet running on the GPU when other DNNs are concurrently run on the DLA of Xavier AGX. The slowdown is calculated based on the standalone GPU execution of GoogleNet where there are no other concurrently running DNNs. *HaX-CoNN* significantly reduces the shared memory contention slowdown in all experiments.

4.5.3 Adapting Optimal Scheduling to Dynamically Changing Workloads

As discussed in Section 4.3.6, we propose D-*HaX-CoNN* to handle dynamic changes to the autonomous CFGs. Its operation is as follows: (1) It starts with an initial best naïve schedule.[1] (2) As the autonomous loop starts executing with the initial schedule, we periodically replace the initial schedule with a better schedule as Z3 progresses. (3) We continue running Z3 until no further improvement is possible.

To demonstrate the effectiveness of D-*HaX-CoNN*, we perform an experiment where dynamic changes in the CFG are simulated by changing three DNN pairs being executed every 10 seconds. DNN pairs are the same within experiments 2, 5, and 1 in Table 4.8, respectively. Figure 4.7 depicts the concurrent execution time of the DNN pairs (*i.e.*, latency per image) as they change. In this experiment, D-*HaX-CoNN* is run on a single CPU core with an initial schedule given by baseline. We update the schedules at 25ms, 100ms, 250ms, 500ms, and 1.5s after starting Z3. The blue lines correspond to the execution time of the updated schedule. The optimal schedule for each pair (represented by a yellow line) is calculated to denote the *oracle* solution that D-*HaX-CoNN* is expected to reach.

[1]We do not start with a Herald or H2H schedule since they also take seconds to return a schedule.

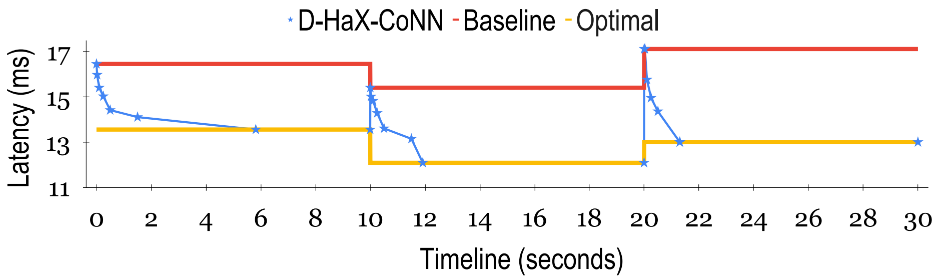


Figure 4.7 A dynamic execution scenario where the target CFG (*i.e.*, DNN pairs) changes every 10 seconds. D-*HaX-CoNN* is shown to gradually improve the execution time as Z3 is asked to update schedules at 25ms, 100ms, 250ms, 500ms, and 1.5s. Blue stars show the update intervals.

Our results show that D-*HaX-CoNN* quickly converges to the optimal solution. In particular, D-*HaX-CoNN* reaches an optimal solution faster for the second and third DNN pairs (1.9s and 1.3s), compared to the first pair (5.8s), since the latter pair has three DNNs and more layer groups. As explained before, a larger number of layer groups results in potentially more transition points to explore, which then increases the time required to explore all transition candidates for the optimized objective function.

Table 4.9 The scheduling overhead (%) of dynamically running the Z3 solver on a CPU core while AlexNet on the DLA is concurrently executed along with other DNNs on the GPU of Xavier Orin.

CaffeNet	DenseNet	GoogleNet	Inc-res-v2	Inception	MobileNet
0.45%	0.89%	1.64%	0.69%	1.64%	1.31%
ResNet18	ResNet52	ResNet101	ResNet152	VGG16	VGG19
0.16%	%0.23%	0.38%	0.71%	1.12%	1.59%

To evaluate the overhead of running Z3 solver along with the concurrent DNN execution, we conduct another experiment where we run AlexNet on DLA along with various DNNs on GPU while Z3 solver runs on a single CPU core of NVIDIA AGX Orin. The results, presented in Table 4.9, show that running the solver on the fly slows down the DNN execution time by no more than 2%. This is attributed to Z3's low memory footprint and Z3 successfully reduces the size of the parameter search space for our targeted problem into a smaller set.

4.5.4 Exhaustive Evaluation with All DNN Pairs

The DNNs that are run concurrently in the experiments presented in Section 4.5.2 were handpicked to reflect the importance of the use cases in each scenario. In this subsection, we conduct a comprehensive evaluation of *HaX-CoNN*, by running every possible DNN pair in our entire DNN set. Since we test every possible pair, the execution times for two concurrent DNNs can significantly differ. We first check the execution time on DLA and GPU for DNN-1 and compare it to DNN-2. Then, to balance out the discrepancy, we increase the number of iterations for the faster DNN. Such scenarios are quite common in multi-sensor systems where two independent sensor data (*i.e.*, camera and radar) are processed concurrently at different frequencies, or where multiple iterations over consecutive data are required to maintain the system's overall accuracy above a threshold. Results of this experiment are given in Table 4.10 as a lower triangular matrix –The upper triangular matrix is symmetric because we are running DNN pairs. The first row of each cell shows the accelerator(s) where the baseline is the fastest for the corresponding objectives. The second row of each cell shows the percentage of improvements that *HaX-CoNN* was able to achieve over the baseline. In this experiment, due to the complexity of the scheduling and because of similar reasons explained in Section 4.5.2, both H2H and Herald mostly result in worse runtimes than the naïve baselines. Key observations from this experiment include:

1. Any pair involving GoogleNet shows improvement since GPU's performance is close to DLA's performance on GoogleNet and *HaX-CoNN* can exploit different transition points where both accelerators are efficient.

2. Overall, *HaX-CoNN* improves the throughput on 35 pairs out of 45 and identifies that GPU-only execution should be applied to the remaining 10 pairs (which are marked as x in Table 4.10), ensuring that *HaX-CoNN* does not underperform. However, experiments involving VGG19 show improvement only in three pairs. The fastest baselines for this DNN are all GPU-only and the execution of VGG19 on DLA is substantially slower than on GPU. Running another DNN on the DLA slows down the entire execution due to high memory contention. When DenseNet or GoogleNet are paired with VGG-19, *HaX-CoNN* shows a slight speed-up since DLA is proportionally faster than the average on the last layer groups in DenseNet and GoogleNet, and in the initial groups of VGG-19.

3. Despite the execution of CaffeNet on DLA being slower compared to GPU; favoring a GPU-only baseline, *HaX-CoNN* is still able to improve performance since CaffeNet is a compute-intensive DNN and does not cause too much contention when paired with other DNNs.

Table 4.10 Comparison among *HaX-CoNN* and the best baseline for DNN pairs running on AGX Orin.

DNNs	1	2	3	4	5	6	7	8	9	10
1-CaffeNet	GPU **1.13**									
2-DenseNet	GPU **1.14**	H2H **1.18**								
3-GoogleNet	GPU **1.06**	D/G **1.18**	GPU **1.22**							
4-Inc-res-v2	GPU **1.08**	GPU **x**	D/G **1.25**	D/G **1.18**						
5-Inception	GPU **1.10**	GPU **1.15**	GPU **1.15**	D/G **1.06**	H2H **1.05**					
6-ResNet18	GPU **x**	D/G **1.14**	G/D **1.13**	D/G **1.32**	GPU **1.19**	GPU **1.23**				
7-ResNet50	GPU **x**	D/G **1.21**	H2H **1.06**	GPU **1.16**	GPU **1.11**	GPU **1.06**	GPU **1.17**			
8-ResNet101	GPU **1.11**	G/D **1.05**	G/D **1.08**	D/G **1.19**	GPU **1.08**	D/G **1.24**	GPU **1.11**	GPU **1.09**		
9-ResNet152	GPU **1.09**	G/D **1.08**	G/D **1.17**	GPU **1.14**	Her. **1.07**	D/G **1.18**	H2H **1.09**	GPU **1.08**	GPU **1.18**	
10-VGG19	GPU **x**	GPU **1.11**	GPU **1.04**	GPU **x**	GPU **x**	GPU **1.08**	GPU **x**	GPU **x**	GPU **x**	GPU **x**

4.6 Conclusion

We propose *HaX-CoNN*, a scheme that maps layers in concurrently executing DNN inference workloads to the accelerators of a heterogeneous SoC. *HaX-CoNN* holistically considers per-layer execution characteristics, shared memory contention, and inter-accelerator transitions while finding optimal schedules. Our experimental results show that *HaX-CoNN* can improve latency up to 32%.

CHAPTER 5
HARNESS: HOLISTIC RESOURCE MANAGEMENT FOR DIVERSELY SCALED EDGE-CLOUD SYSTEMS

In Chapters 3 and 4, we have maximized the computational capability within an edge device. For distributed systems, computing resources in each tier, such as processing units inside in-the-field edge devices and high-performance servers in clouds, are handled in isolation due to scalability and resource segregation in several domains, such as federated learning, AR/VR platforms, and smart buildings. In this chapter, we will introduce a holistic approach,*HARNESS*. *HARNESS* is a resource management framework that captures diverse computational characteristics of edge-cloud systems with arbitrary topologies and to efficiently manage computational resources across the whole continuum in the scope.

5.1 Introduction

Computing in-the-field (*i.e.*, on the *edge*) is becoming more demanding. The increasing need for on-device intelligence results in a wider deployment of system on chips (SoC), embedding a variety of processing units (PU) and domain-specific accelerators (DSA), to efficiently run applications with minimal power and/or latency [210, 211]. Often, edge devices also rely on more powerful servers to store the data they collect [212], offload some of their computation [213] or synchronize with other devices [22]. In many emerging domains such as federated learning, autonomous systems, and AR/VR systems, the execution of the workloads spans numerous *nodes* with varying degrees of computational power (*i.e.*, heterogeneous), including *edge devices* and *servers* (or “cloud").

Over the last two decades, resource management for HPC systems [110, 111, 214] and data centers [116, 117, 215] has been broadly studied. In the meantime, edge platforms became more accelerator-rich, powerful, and efficient [35]. However, as the execution of emerging workloads evolved to span computing systems with vastly heterogeneous computational tiers, which we refer to as *diversely scaled edge-cloud systems* (DECSs), existing approaches fail to provide a *scalable resource modeling and management solution* that can *holistically* capture the *edge-cloud system*'s performance *accurately* and *adaptively* (see §2.2 for details). Specifically, DECSs have the following key characteristics which present unique challenges when considered together: (1) *High degree of computational heterogeneity*: Computational nodes (*e.g.*, edge devices and cloud servers) in a DECS constitute a multi-tiered computing topology. The nodes in each tier embed multiple PUs that have significantly diverse computational capabilities compared to the other nodes in the same and different tiers. (2) *Segregated/isolated computational environments*: Computational resources are segregated into various clusters to abstract or group nodes based on their functionality or isolation requirements (*e.g.*, mesh-IoT [216], function-as-a-service [217] and cloud gaming [218]). For example, the internal architecture and operation of the servers in the cloud are often invisible to edge devices and vice versa. (3) *Dynamically changing computational capabilities*: The performance of the system varies over time depending on the newly added or removed nodes and the slowdown caused by shared resource usages across various tiers.

In this paper, we propose *HARNESS*, **H**olistic and **A**daptive **R**esource ma**N**agement for diversely scaled heterogeneous **E**dge-Cloud **S**y**S**tems. *HARNESS* leverages a multi-layer *graph-based HW representation* to enable flexible and scalable abstractions of the computational resources in DECSs. The graph-based HW approach champions a modular strategy, allowing the incorporation of existing performance models, while still capturing interactions at higher levels across the system. *HARNESS* deploys a multi-tiered *Orchestrator* (*ORC*) mechanism to hierarchically locate resources a task could be mapped to, while taking the slowdown caused by shared resource usages at different tiers into account. *ORC*s enable scalable resource management across isolated/segregated computational node clusters without the need for any group to have complete performance models and task-assignment knowledge of the other group. *ORC*s utilize an internal *Traverser* mechanism to predict the shared resource slowdown for a given mapping between a set of tasks and target PUs. Our slowdown modeling approach uniquely decouples the standalone performance of a component and the slowdown caused by shared resource usage; hence, simplifying the performance modeling and increasing the prediction accuracy.

Overall, *HARNESS*, to the best of our knowledge, is the first framework to enable a *holistic resource management* approach utilizing *all computational resources* in heterogeneous edge-cloud-based systems while taking *multi-tier interference* and *dynamic HW topology changes* into account.

5.1.1 Contributions

Our work makes the following contributions:

- We present a *graph-based multi-layer HW representation* scheme that is capable of expressing *arbitrary topologies* of HW components and their *interactions* in DECSs.
- We devise a *Traverser* logic to automate the process of predicting the performance of a given set of tasks on a target set of PUs while also accounting for the *shared resource slowdown* among concurrent tasks.
- We design a multi-tiered, decentralized *Orchestrator* mechanism that scalably finds a mapping of a task to a local or remote PU while satisfying the task's constraints. Our proposed *ORC* mechanism uniquely supports *segregated resource clusters*, which is common in DECSs.
- We demonstrate the utility of *HARNESS* on DECSs from two different disciplines that we deploy in the field. Our experiments show a latency improvement of up to 47% and a reduction in the prediction error

rate from 27.4% to 3.2% with less than 2% scheduling overhead.

The remainder of this chapter is organized as follows: Section 5.2 provides background and motivation for holistic edge–cloud resource management. Section 5.3 details HARNESS itself—design requirements, the overall architecture, *HW-GRAPH*, *Traverser*, and *Orchestrator* components. Section 5.4 describes the experimental setup, including target applications and system configurations. Section 5.5 evaluates HARNESS in terms of performance, model accuracy, adaptability, scalability, overheads, and mapping strategies.

5.2 Motivational Study and Related Work

Motivation: To demonstrate the need for a comprehensive resource manager for DECSs, we have developed and studied the behavior of a real-life edge-cloud application. The application implements a remotely rendered 3D virtual reality (VR) environment, where users wear untethered VR glasses equipped with multi-accelerator SoCs. Most of the tasks on the VR glass (*i.e.*, edge device) could run locally, but a powerful remote server is still needed to deliver a low-latency and high-definition rendering of the 3D environment which VR users navigate using their body movements. To minimize end-to-end latency, we deploy an RNN-based predictive rendering approach [219] to speculatively predict the future poses of the VR user based on earlier body movements. We explore the execution of this VR application on a mini edge-cloud platform consisting of NVIDIA's Orin series edge SoCs and a remote server with a discrete GPU. In addition to CPU and GPU, Orin SoCs employ a special accelerator named Video Image Compositor (VIC), which can efficiently run some computer vision functions. The left part of Figure 5.1 shows the linear order of five tasks of interest. Each task can be run on different set of PUs (GPU, VIC, CPU) on the server and edge devices. The upper-right part of Figure 5.1 shows the frame execution pipeline when two edge devices share a single server for remote rendering operations. Initially, only one edge device (e1) is active with another edge device (e2) joining later during execution. Eight PUs (C, G, V for edge devices and C, G for server) work together to complete the five tasks for each frame captured by the edge devices.

We derive several key insights from Figure 5.1: (i) Pose estimation is initially run on server GPU because it is faster, including the communication latency. However, since the server GPU is prioritized for rendering tasks, *T1:pose-estimation* (lilac) is swapped from the server to the edge after the first frame. *This decision requires rapidly and scalably checking dynamic PU availability at every task assignment.* (ii) As new types of tasks start executing in the system, the mapping of older tasks could be changed depending on latency requirements. For example, the CPU of edge device one was executing *T3:decode* (green) tasks only, until *T5:display* (pink) tasks started arriving. The best decision once this occurs is to offload *T3:decode* tasks to the GPU, despite the GPU already handling *T1:pose-estimation*. *This decision requires heterogeneity, latency, and contention-aware mapping decisions.* (iii) New edge devices joining the system may cause a slowdown on the server due to multi-tenancy.For example, towards the end, two *T2:render* (turquoise) tasks begin arriving from both edge devices. The best mapping decision is to keep processing them on the server, despite the slowdown, since the latency requirements of the edge devices are still met. *This decision requires multi-tiered consideration of shared resource slowdown and adaptive task mapping w.r.t. the changing HW.*

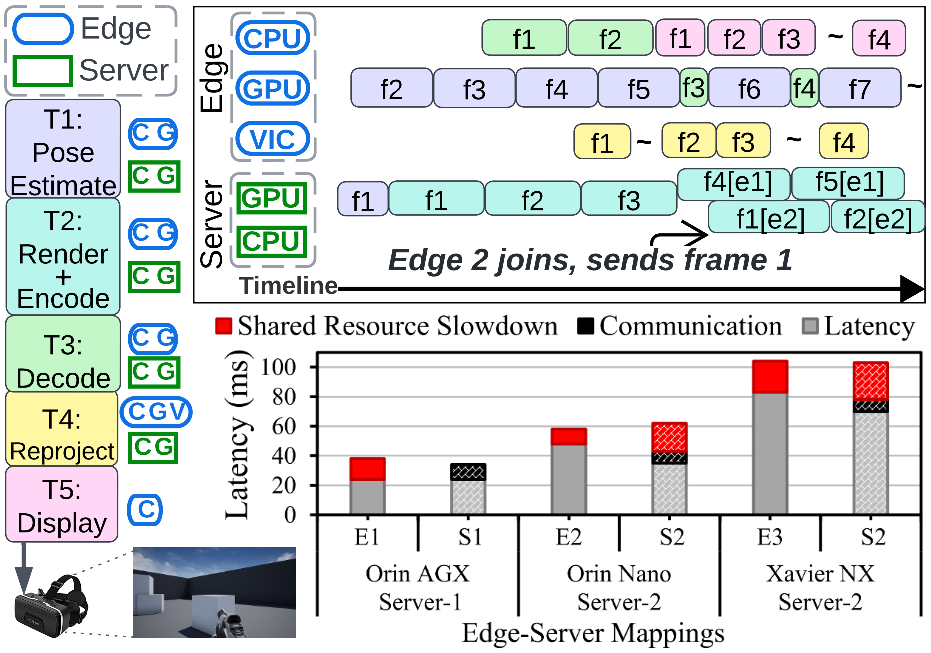


Figure 5.1 *Left:* The tasks in VR app and PUs they could run on. *Upper:* The frame pipeline for each PU. f4[e1] refers to frame 4 on edge device 1. *Lower:* [E]dge and [S]erver side latency, communication, and slowdown breakdown for VR application on a DECS containing 3 edge devices and 2 servers.

Bringing the example one step further, suppose there are two slower edge devices (Orin Nano *E1* and Xavier NX *E2*) and one faster edge device (Orin AGX *E3*), all of which share two servers (*S1* and *S2*) for the remote rendering task. The bottom portion of Figure 5.1 gives a breakdown of the time spent to process a single frame on each edge-server pair. While mapping the rendering tasks, the resource manager can identify that there are lenient latency constraints on the slower edge devices, since they will process other tasks in the pipeline slower. As a result, mapping their rendering tasks to the same server will not violate their latency requirements, although the collocated tasks will run longer due to shared resource use. On the other hand, this mapping will enable the other server to handle the stricter latency requirement of the faster edge device allowing the latency requirements for all rendering tasks from all edge devices to be met. The ability to make such a decision requires a comprehensive knowledge of all computational devices and task mappings in the entire system. However, due to resource segregation, neither edge devices nor the servers will have such knowledge. *This observation highlights the need for a holistic, multi-layer, contention-aware and adaptive resource manager that could operate under resource segregation.*

Table 5.1 Feature comparison against the state-of-the-art.

	ACE [21]	Borg [116, 117]	LaTS [22]	PARTIES [85]	ASPEN [220]	IRIS [80, 81]	CloudVR [221]	Hwloc [104]	EdgeMtrx[123]	***HARNESS***
Arbitrary hardware topologies	✓	✓	✗	✗	✗	✓	✗	✓	✗	✓
Scalable resource management	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓
Not application-specific	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓
Shared resource slowdown	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓
Dynamic adaptability	✗	✓	✓	✓	✗	✓	✓	✗	✓	✓
Heterogeneous PUs in a node	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓
Inter-node heterogeneity	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓
Isolated resource management	✗	✗	✗	✓	✗	✗	✗	✗	✓	✓

Gap in the literature: Table 5.1 provides an overall comparison between *HARNESS* and other relevant works by the features they support: (i) the ability to represent arbitrary HW configurations and topologies within and across nodes, (ii) scalable resource management across nodes, (iii) support for arbitrary class of applications (*i.e.*, not application-specific), (iv) accounting for the slowdown caused by shared resource usage, (v) dynamically adapting to HW changes, (vi) modeling the performance of heterogeneous processors within a node, (vii) modeling diverse heterogeneity across the nodes, and (viii) supporting segregated/isolated resource clusters. Overall, *HARNESS* is the only work that supports all these features. Such comprehensive support is necessary to scalably and accurately model and manage resources in DECSs. The three studies (*LaTS* [22], *ACE* [21], *Multi-tier CloudVR* [221]) that we compare our work against are further explained in §5.4.2.

5.3 *HARNESS*: A Holistic Resource Manager for Diverse Edge-Cloud Systems

This section introduces HARNESS, a holistic, contention-aware resource manager that unifies heterogeneous edge-to-cloud resources through a multi-layer hardware graph, online slowdown modeling, and hierarchical orchestrators. We first list the design requirements of diversely scaled edge-cloud systems, then show how HARNESS's *HW-GRAPH*, *Traverser*, and *ORC* mechanisms collectively satisfy them.

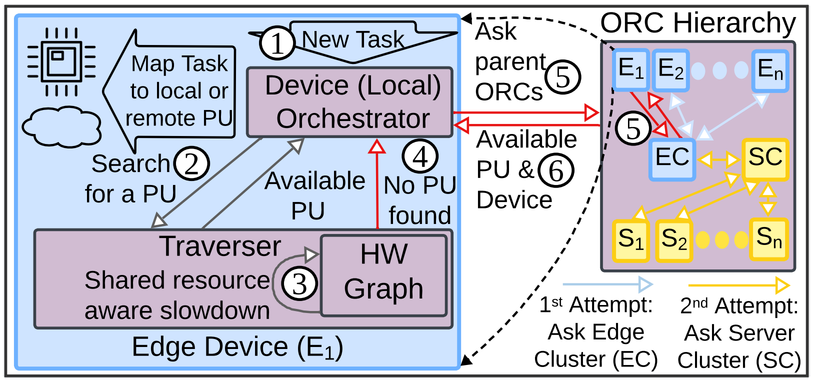

New Task
1
Map Task to local or remote PU
Device (Local) Orchestrator
Search for a PU
2
Available PU
4
No PU found
Traverser
Shared resource aware slowdown
3
HW Graph
Edge Device (E1)
Ask parent ORCs
5
Available PU &
6
Device
ORC Hierarchy
E1
E2
En
EC
SC
S1
S2
Sn
1st Attempt: Ask Edge Cluster (EC)
2nd Attempt: Ask Server Cluster (SC)

Figure 5.2 An overview of the *HARNESS* framework

5.3.1 Design Requirements

To address the unique resource management needs of DECSs, we identify and target the following requirements while designing our proposed approach: (1) Computing systems with arbitrary and abstract topologies should be supported, (2) resource management should be de-centralized and support resource abstraction and segregation, (3) assignment of a task to a PU should be scalable and adaptive to dynamic changes in DECS, (4) the cumulative slowdown due to shared resource contention at different levels should be taken into account, and (5) modular integration of various performance and slowdown prediction models should be supported. Through the design of *HARNESS*, requirements (1) and (5) are captured by the graph-based representation (§5.3.3), requirements (2) and (3) are addressed by the *Orchestrators* (§5.3.5), and requirement (4) is satisfied by the *Traversers* (§5.3.4).

5.3.2 Overview of *HARNESS*

HARNESS is composed of three major components:

- *HW-GRAPH*: A multi-layer graph-based HW representation that models the connections and interactions between the computational nodes and PUs in a DECS (§5.3.3).
- *traverser*: A mechanism to automate the performance and slowdown prediction by traversing the tasks in a control flow graph (CFG) against the *HW-GRAPH* of a computational node (§5.3.4).
- *Orchestrators (ORCs)*: Hierarchically organized, per-node daemons that facilitate the assignment of tasks to PUs in a decentralized manner (§5.3.5).

An overview of *HARNESS* is depicted in Figure 5.2. *HARNESS* fosters a decentralized and edge-triggered resource management scheme. Edge devices may run the same or different applications and these applications are assumed to be composed of tasks each corresponding to a unit of assignable computation (*e.g.*, kernel). The system developer/programmer, *i.e.*, the user of *HARNESS*, is expected to utilize the resource manager for the tasks representing computationally significant and acceleratable code blocks. Such tasks are to be labeled as `TASK`s (see *Application representation* paragraph in §5.3.3) by the developer. Whenever there is a `TASK` or a set of `TASK`s that are ready to execute, the developer invokes our resource manager via the `MapTask()` API call. These `TASK`(s) are assumed to be either created directly or freed through dependency resolution. The developer passes the set of *TASKS*s, any dependencies, per-task constraints, and the overall objective (such as minimizing the overall latency). The developer controls the granularity of the mapping by passing the number of desired `TASK`s to schedule, and *HARNESS* decides which node (*i.e.*, PU on device) to map each `TASK` onto. Overall, the steps *HARNESS* utilizes during operation are as follows: ① An *ORC* is associated with each higher-level node (*e.g.*, an edge device, a server or an abstracted/isolated node group) in the *HW-GRAPH* and run as a daemon. The `MapTask()` function is called on the local *ORC* to locate the resource(s) that the `TASK`(s) supplied by the developer can be mapped onto. ② The local *ORC* first invokes its internal *Traverser* to see whether any PU(s) in the same node could run the `TASK`(s) under given constraints. ③ *Traverser* utilizes the *HW-GRAPH* to query each PU in that device for its current `TASK`(s) and predicts the performance and potential slowdown if there are concurrently running `TASK`(s) sharing the same resources with that PU. ④ If the local device does not have a suitable PU that can execute the `TASK` under the given constraints, the local *ORC* requests resources from its parent *ORC*. ⑤ This triggers a hierarchical query across other *ORC*s—first among the edge cluster(s) and then, if needed, among the server cluster(s)—each invoking its respective *Traverser* to locate an appropriate PU. ⑥ Based on the returned resource/PU type (*e.g.*, CPU, GPU, VIC), the developer then invokes the proper implementation of the `TASK` to run on the designated resource/PU.

5.3.3 *HW-GRAPH*

The foundations of *HARNESS* are built upon a graph-based representation scheme, *HW-GRAPH*, customized to model the HW components and their complex interactions in a DECS. *HW-GRAPH* relies on a connected multi-layer graph topology [222, 223] to describe the hierarchical interactions between multiple levels of HW abstractions. In *HW-GRAPH*, a node corresponds to one of these: (i) a PU, such as a CPU core or GPU, (ii) a storage unit, such as cache or memory, (iii) a dedicated controller circuit, such as memory controller or network switch, (iv) an abstract component, whose internals are not known, or (v) a sub-graph representing a high-level component which groups smaller components, such as a CPU with multiple cores and caches or segregated resource clusters (*e.g.*, ‘the cloud’ composed of multiple servers). The edges in the *HW-GRAPH* correspond to interconnects linking the nodes listed above. Edges define (i) how PUs are connected to each other and the memory, and (ii) how higher-level nodes (i.e., edge devices or servers) are connected to each other in the network. The graph-based representation of the entire continuum enables *HARNESS* to *algorithmically* (*i.e.*, in a generalized and automated way) (a) traverse the PUs in an SoC or server, (b) locate the storage (*e.g.*, memory) and control components that two PUs share as they operate, (c) virtually group sets of computational devices (such as various edge or cloud clusters) for scalability, and (d) hierarchically identify other nodes in a DECS that a given node can offload its computation onto. Each node stores its own *HW-GRAPH* to represent its sub-components. All HW changes in the system (e.g., device additions/removals, network bandwidth changes) are captured via *HW-GRAPH* updates and propagated to the corresponding parent and child *ORC*s during execution. These features of *HW-GRAPH* enable seamless operation of *Traverser* and *Orchestrator* mechanisms, described later in §5.3.4 and §5.3.5, and let *HARNESS* support DECSs with any arbitrary computational and communicational topology.

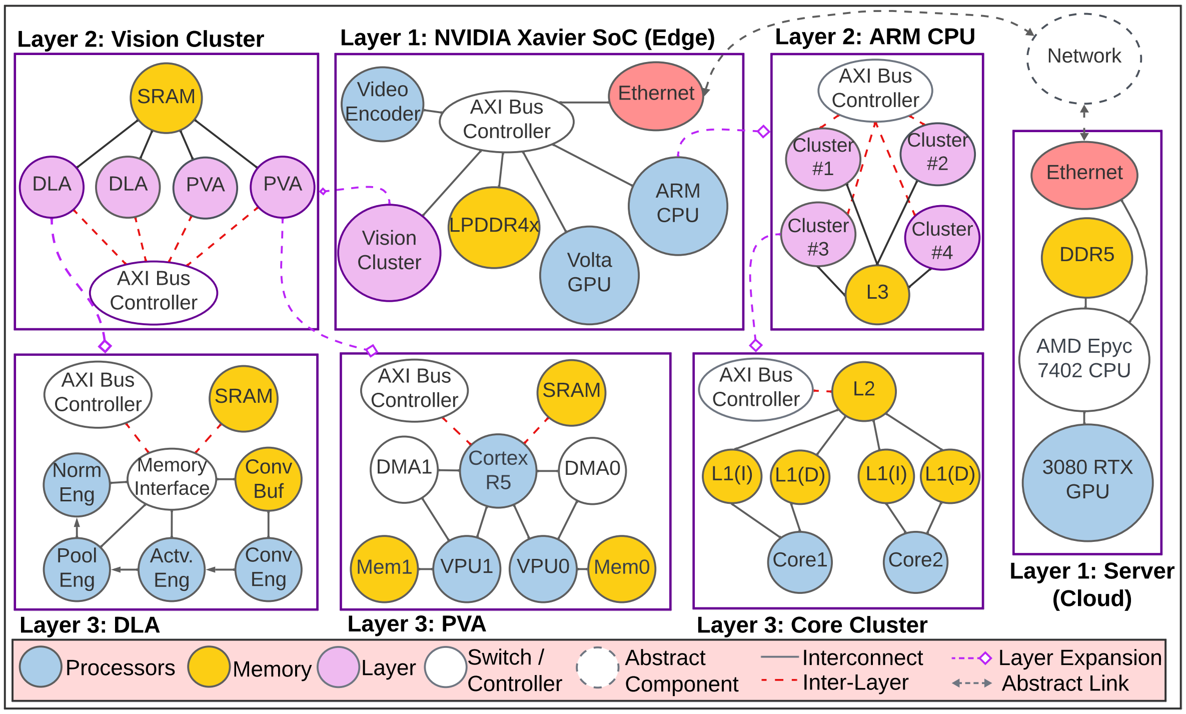


Figure 5.3 An example *HW-GRAPH* for a DECS made up of an NVIDIA Xavier SoC and a server with discrete GPU connected over a network.

Figure 5.3 illustrates a *HW-GRAPH* representation of an edge device, NVIDIA AGX Xavier SoC, connected to a server in the cloud, similar to the VR application given in §5.2. In this example, the edge device and server are the top-most layers in the graph (*i.e.*, layer 1), and they are connected via abstract components and links, which correspond to the unknown network infrastructure. The level of component detail increases in layers 2 and 3, and dashed connections across layers represent the relationship between abstracted and detailed versions of components. In the given example, *HW-GRAPH* can be used to automatically uncover a relation between `DLA` (deep learning accelerator) and `PVA` (programmable vision accelerator) residing under Layer 2 Vision Cluster. Concurrent execution on these DSAs results in the shared usage of multiple components: `SRAM` in `Vision Cluster` and `LPDDR4x` (*i.e.*, shared system memory). Intersection of the compute/memory paths used by two `TASK`s concurrently running on `DLA` and `PVA` algorithmically unveils these shared resources. While the example given in Figure 5.3 is a highly detailed representation of the NVIDIA Xavier SoC, the developer may choose to represent only the relevant components of the system, such as the `CPU, GPU, DLA, PVA`, and the shared `LPDDR4x`. With these components, the developer could still model shared memory contention between the PUs, but the L2 cache based contention across the CPU cores would not be modeled if layer 3 is omitted.

The API: The *HW-GRAPH* corresponding to a DECS system is created via an object-oriented interface. Every HW component derives from either `Node` and `Edge` objects. `TASK`(s) can only be mapped to higher-level `Node`s in the *HW-GRAPH*. For example, the `Convolution Engine` sub-component of the `DLA` in Figure 5.3 cannot run a computational task alone, but the `DLA` can. Such higher-level `Node`s, *i.e.*, PUs, extends the `Predictable` interface and implement the `predict()` function so that the time it will take for the PU to run a specific task can be queried *algorithmically* by the Traverser (as explained in §5.3.4).

Performance prediction: The `predict()` function takes a `TASK` object as input to retrieve its previously modeled performance data for the PU-`TASK` pair. This function is designed in a *modular way* to support existing component-level performance prediction mechanisms, such as empirical profiling [224], Roofline [225], and analytical modeling [220]. To construct the *HW-GRAPH*, the developer creates `Node` and `Edge` objects that the target system is composed of. *HARNESS* ships with *HW-GRAPH* templates for commodity edge platforms and server components so that the developers can easily build upon. Automated creation of *HW-GRAPH* is left as future work. Each `Predictable` component implements `getComputePath()` function, which returns the shortest path between the PU and the other memory/control sources the `TASK` relies on. Such a list of resources is obtained during profiling and stored inside the `TASK` object. Traverser calls the `getComputePath()` function to automatically identify shared resources and account for any potential slowdown. In our experiments, we pre-profile the `TASK`s to be executed for all possible target PUs. While profiling `TASK`s beforehand may not be possible for a wider range of DECSs and applications, many DECSs run a pre-determined set of `TASK`s in a repetitive manner [10, 226]. Therefore, in this work, *we concentrate our efforts on resource management only* while also *enabling developers to modularly integrate their performance modeling methodology* via the `predict()` interface.

Application representation: We assume each device executes the same or different applications that are composed of computational regions (*e.g.*, kernels, API calls etc.) performing a specific function (*e.g.*, convolution, matrix multiplication etc.). Right before such function calls or code regions, the users of *HARNESS* (*i.e.*, application developers) are expected to create a `TASK` object that is composed of an identifier, input/output sizes, task constraints (*e.g.*, deadlines) and the list of local and remote PUs that the block could be executed on. *HARNESS* assumes that only computationally significant regions of the application will be denoted as `TASK`s and the remaining code will be executed locally. The user then invokes the *HARNESS* resource manager (explained in §5.3.5) via the `MapTask()` API call and passes the `TASK` object as a parameter. At this point, the computational region corresponding to the `TASK` object is assumed to be ready to execute (*i.e.*, all dependencies are resolved) and will be mapped to the PU returned by the `MapTask()` call. If there are multiple `TASK`s ready to execute, then the developer may pass a set of `TASK`s, their dependencies (if any) and per-task constraints as parameters. The ability to pass one or more `TASK` objects at once enables the developer to control the granularity of the mapping – the trade-off between task mapping overhead and task granularity is evaluated in §5.5.5. Once the `MapTask()` function returns a PU (local or remote) that satisfies the `TASK` constraints, the developer is expected to execute or offload the corresponding task region accordingly.

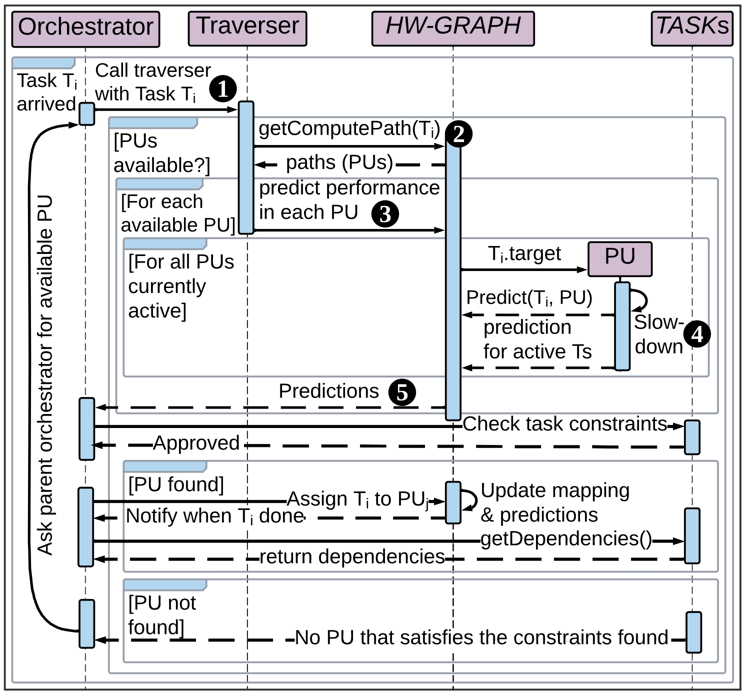


Figure 5.4 The internal operation of *Traverser* and how *Orchestrator*s utilize *Traverser*s.

5.3.4 Traverser

We devise a *Traverser* as the mechanism to automate the process of predicting the performance of a given set of `TASK`s and their dependencies on a target set of PUs. *Traverser* acts as a *cost function* used by *ORC*s and it accounts for the potential shared resource slowdown among concurrently running `TASK`s while calculating the cost (*i.e.*, performance). *Traverser* uniquely automates this process by "traversing" the sub-components of a higher level component in the *HW-GRAPH* at each task assignment. *Traverser* is invoked by the *ORC* (see §5.3.5 for details) to predict the performance of a `TASK` on a specific PU to check whether a mapping between them meets constraints without violating the constraints of existing tasks. *traverser*, whose sequence diagram is depicted in Figure 5.4, operates as follows: ① Starting from the independent `TASK`(s) in the tasks set provided by the developer, it traverses through the dependencies in a time-ordered fashion by following the parallel & serial regions of the CFG and dependencies. ② Traverser honors the task-to-PU assignments, which are provided by *ORC*. ③ Traverser initially calls the `predict()` function to find standalone execution time on the PU in which a task is mapped. ④ Then, after identifying the contention intervals, as prescribed below, *Traverser* calls the `slowdown()` function with the collocated `TASK` information so that the slowdown is accounted for in the initially predicted performance and ⑥ returned to the *ORC*.

Contention intervals: When multiple `TASK`s run simultaneously on different PUs, `TASK`s will be slowed down non-uniformly throughout their execution depending on which other `TASK`s are running on the same computational node or PU at that particular time. Figure 5.5 illustrates this behavior by depicting execution timelines of three workloads with five hypothetical `TASK`s that are running to completion on three PUs. To breakdown the slowdown calculation, we divide the initial predicted execution timeline (with no slowdowns) into *contention intervals*. Intervals are separated by time makers (*e.g.*, t_0 to t_4) each indicating the beginning or end of a task. This results in each interval to contain at most one task for each PU, hence simplifying the slowdown calculation. Then, the amount of slowdown for each interval is quantified iteratively as explained below.

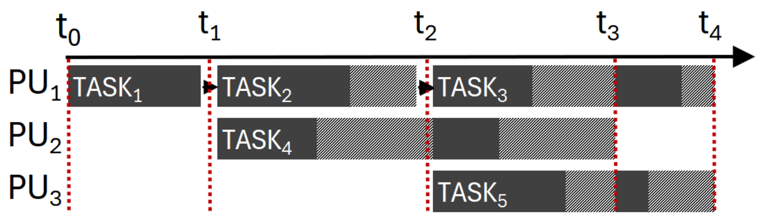


Figure 5.5 Timeline of five tasks completed on three PUs, with shaded areas showing additional slowdown and dashed lines marking *contention intervals*.

Slowdown calculation: HARNESS uniquely decouples the calculation of slowdown from standalone performance models: (1) Only once for each system, the resources that can be shared are characterized and profiled for the slowdown they will experience per the amount of concurrent use they encounter. (2) Then, for a given resource, such as memory or a PU, each task is identified by the generalized amount of usage for that specific resource, such as requested memory throughput or core utilization, respectively. (3) Finally, during runtime, the `predict()` function uses concurrent `TASK`s' amount of usage for that specific resource, and `slowdown()` function incorporates corresponding slowdown using the models built in the first step.

It is essential to note that the novelty of our slowdown calculation comes from the modular and decoupled integration methodology that scales across multiple tiers of DECS. The development of per-PU slowdown models is outside of the scope of this work, and existing slowdown models are utilized in our experiments.

Handling prediction inaccuracies: Since *HARNESS* relies on the performance models provided by the user, the predictions of execution times could be inaccurate. To accommodate predictions that are consistently off within margins, we integrate a rolling-window-based correction mechanism. *Traverser* achieves this by comparing the predicted execution times for the specific *TASK* and PU pair at hand with the actual time it takes to execute the *TASK*. If the difference is above a predetermined threshold θ, then *Traverser* calculates a correction factor σ by averaging the latest n predictions for the specific *TASK* and PU pair. Instead of relying solely on the predicted execution time τ calculated by the `predict()` and `slowdown()` functions, *Traverser* uses $\tau \times \sigma$ for upcoming task invocations. In our experiments, θ is set to 5%, which is a marginally higher value than the average model error rate that we measured in §5.5.2. Also, we find that a window length of $n = 15$ predictions is sufficient to capture the dynamic changes in the system that affect prediction accuracy.

5.3.5 Orchestrator

The *ORC* mechanism is an integral component of *HARNESS* and facilitates assignments of tasks to PUs in a scalable way. To achieve this, *ORC*s work in a hierarchical tree topology and *ORC*s are responsible for finding an appropriate PU to map a given task to. An *ORC* daemon is created for every non-leaf computational node (*e.g.*, edge device or a server) of *HW-GRAPH*. They internally utilize *Traverser* and *HW-GRAPH* to look for PUs in their hierarchy. Each *ORC* could only communicate with its parent and child *ORC*s, and none of the *ORC*s in the system is assumed to have the full knowledge of the *HW-GRAPH* or other *ORC*s in the system. Figure 5.6 depicts an *ORC* hierarchy for the example DECS we described in §5.2. There is an *ORC* associated with each edge device and server (2, 3, 4, 6, and 7). In addition, there are higher-level *ORC*s (1 and 5) associated with virtual abstractions created for edge and server clusters. There is also a *Root* level *ORC* that is only known by *ORC*s 1 and 5. There are no *ORC*s associated with the leaf-level nodes (*e.g.*, the PUs corresponding to node #3, #5, #6 and #7). This is because the parent *ORC*s (2, 3, 4, 6, and 7) are assumed to have full knowledge of the PUs that are immediate children of the node (*e.g.*, Node #2 corresponding to Edge #1) that they are overseeing.

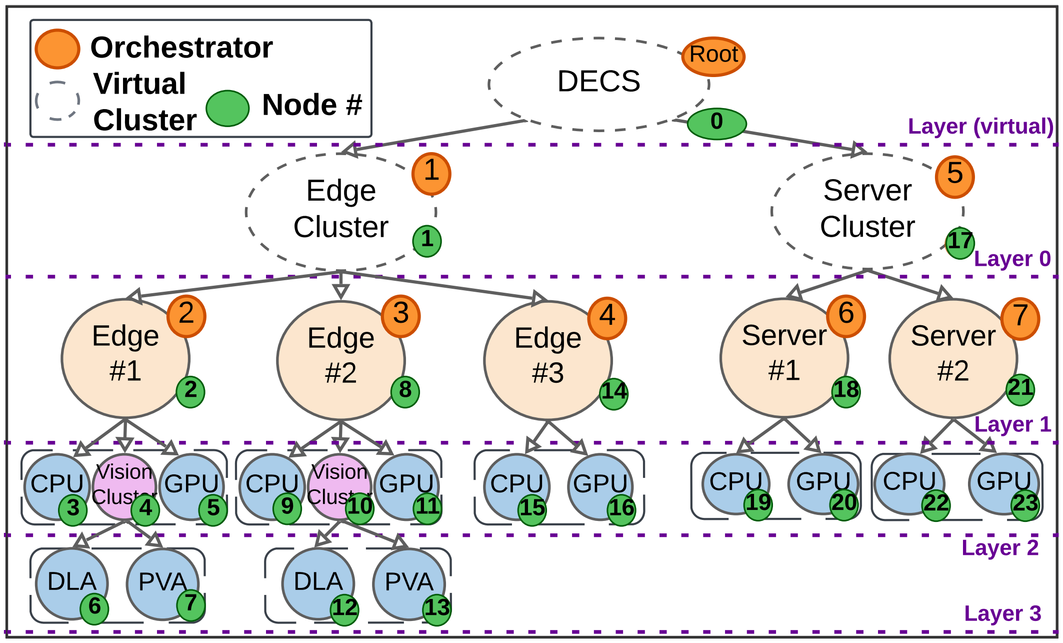


Figure 5.6 *ORC* hierarchy on a DECS with three edge devices and two servers.

A pseudo algorithm for how *ORC*s work is given in Alg. 1. The working principles of the *ORC* mechanism are as follows: **(i)** For each new `TASK` T_i (along with its constraints C_i and dependencies TD_i), an edge device invokes the `MapTask()` function of its local *ORC* (line 3). **(ii)** The local *ORC* iterates over its children (line 19): (a) If the child is a leaf node N_j (*i.e.*, a PU that a `TASK` could be directly assigned) (line 20), then the *ORC* invokes *Traverser* on the PU to get a performance prediction that also accounts for the slowdown in the new `TASK` (line 11) and actively running `TASK`s (line 15). (b) If the child is an *ORC*, `MapTask()` is recursively invoked on the child *ORC* (line 23). (c) If a child PU that satisfies the `TASK`'s constraints is found, then that PU is returned. (line 7). **(iii)** Otherwise, the search is propagated to the parent *ORC* (line 8): (a) Parent *ORC* invokes `MapTask()` for the siblings of the local *ORC* (line 28). (b) If a suitable PU is not found, the search is propagated to other *ORC*s in a depth first search order.

To determine whether a suitable remote PU is found, the latency to communicate with such PUs from the origin PU is also factored in while checking for the constraints. To prevent deadlocks, when a remote *ORC* finds a suitable PU, that PU is provisionally reserved. The reservation is released if the originating *ORC* does not assign the `TASK` or timeouts. Once `MapTask()` returns, the user initiates the task execution via `ExecuteTask()` function. For local PUs, the task is executed locally by passing the input to the previously compiled binary directly. For PUs on other nodes, *HARNESS* internally invokes remote task execution.

Algorithm 1 Task Allocation via Orchestrator Mechanism

```
1: **Input:** $TaskId$, $Dependencies$, $Constraints$, $Objective$,
2: **Output:** $targetNode$
3: **Function** MapTask($T_i, N_j, C_i, TD_i$)
4:     **for all** $Child \in$ ChildrenOf($N_j$) **do**
5:         Add AskChildren($Child, T_i, C_i, TD_i$) to $BestNodes$
6:     **if** $BestNodes$ is not empty **then**
7:         BestNode ← select best in $BestNodes$ on $Objective$
8:     **else** BestNode ← AskParent($N_j, T_i, C_i, TD_i$)                ▷ Find new nodes
9:     **return** BestNode, Result
10: **Function** CheckTaskConstraints($T_i, N_j, C_i, TD_i$)
11:     result ← InvokeTraverser($T_i, N_j, TD_i$)                ▷ Predict slowdown for $T_i$
12:     **if** not SatisfyConstraints($N_j, T_i, C_i$) **then**
13:         **return** result, False                ▷ $T_i$'s constraint is failed
14:     **for all** activeTask *in* $N_j$ **do**                ▷ predict slowdown on active Tasks
15:         InvokeTraverser($activeTask, N_j, TD_i$)
16:         **if** not SatisfyConstraints($N_j, activeTask, C_i$) **then**
17:             **return** result, False                ▷ one active task's const. failed
18:     **return** result, True                ▷ all constraints OK
19: **Function** AskChildren($N_j, T_i, C_i, TD_i$)
20:     **if** IsLeaf($N_j$) **then**
21:         **if** result ← CheckTaskConstraints($T_i, N_j, C_i, TD_i$) **then**
22:             **return** result, $N_j$                ▷ Task can be assigned
23:     **else** **return** MapTask($T_i, N_j, C_i, TD_i$), $N_j$                ▷ Ask children
24: **Function** AskParent($N_j, T_i, C_i, TD_i$)
25:     parent ← ParentOf($N_j$)
26:     **for all** child $\in$ ChildrenOf(parent) **do**                ▷ ask each child of parent
27:         **if** child $\neq N_j$ **then**
28:             **return** MapTask($T_i$, child, $C_i, TD_i$), $N_j$
```

Remote task executions are carried out similarly to the serverless computing paradigm outlined in [227]. `ExecuteTask()` serializes the *TASK*'s inputs and transfers the data to the remote node through the communication channel established between the local and remote ORCs. The binary required for the remote *TASK* and the PU pair is assumed to be previously initialized by the user via the `registerBinary()` function (see §5.3.6 for details). Once remote execution is invoked, the remote ORC daemon service writes the input data to a predetermined directory expected by the *TASK* binary and initiates execution. Once the execution is complete, the output is streamed back to the local ORC initiating the *TASK* execution. For both remote and local *TASK* executions, the corresponding *TASK* binaries are executed directly on the remote system, without any containerization; as we solely focus on system utilization in this work and not the isolation of resources.

5.3.6 Usage of *HARNESS*

HARNESS framework provides the following:

- *HW-GRAPH* API for users to build HW-GRAPH models of their targeted compute nodes.
- Implementations of ORC and Traverser mechanisms and interfaces for users to interact with local ORCs.
- Daemon services, communication and synchronization between ORCs at different compute nodes.
- Local and remote task execution mechanisms, including input/output data serialization and transfer for remote tasks (See the end of §5.3.5 for details).
- A collection of *HW-GRAPH* models for commodity mobile and autonomous SoCs.

Users of *HARNESS* are expected to:

- Create *HW-GRAPH* representation using API calls or the pre-made models shipped with *HARNESS*.
- Create *TASK* objects for the kernel executions that will be managed by *HARNESS*
- Declare `predict()` and `getComputePath()` functions for the PUs and *TAS*Ks that the PUs in the system could run.
- Provide and register device-specific kernel implementations, *i.e.*, binaries, for each TASK.
- Call the `MapTask()` function to find a target PU for a TASK and call `ExecuteTask()` to run the TASK.

Listing 5.1 provides code snippets for an example use of *HARNESS* from CloudVR application. The example shows the steps that the user needs to follow to add the *HW-GRAPH* for the Jetson Xavier AGX hardware, and then construct, map, and execute a "Reproject" task. First, the user adds a XavierSoC node to *HW-GRAPH*, indicating key components such as ARM architecture, accelerators (*i.e.*, GPU, DLA, PVA), and memory resources. Subnodes (ARMCluster, DLA_Cluster, PVA_Cluster, DRAM) in layer 2 are then added to represent additional details like caches, multiple accelerator cores, and shared memory. Next, a new `TASK` object is created with a latency constraint of 30 ms. Then, the user calls `registerBinary()` to let *HARNESS* know which binary a given *TASK* and PU pair should use. The user then implements `predict()` and `getComputePath()` functions for different PU targets that the *TASK* could run on. The steps described so far are performed only once for the underlying hardware (*i.e.*, *HW-GRAPH* related operations) or for each *TASK* type (*i.e.*, *TASK* related operations). During runtime, when it is time to launch the "Reproject" task in the application, the user calls `MapTask()` function to ask its local ORC for a suitable PU to execute the task at hand. If multiple tasks are asked to be mapped at once, then the user could specify an optional "dependency" parameter for the *ORC*s and *Traverser*s to take the dependencies into account when mapping the tasks. Finally the user calls the `ExecuteTask()` function

Listing 5.1: Example usage of HARNESS APIs

```
import harness

hw_graph = harness.HWGraph.init()
orc = harness.ORC.init("ID", parent="Edge Cluster")

# An example of how one node is added in Layer 1
hw_graph.add_node("XavierSoC", harness.NodeSpec(arch="arm64", CPU="ARMCluster", GPU="VoltaArch",
       VisionCluster="2xDLA_2xPVA", VideoEncoder="H264", mem="DRAM"), parent="Edge Cluster")

# Layer 2 sub-nodes under Xavier SoC
hw_graph.add_node("ARMCPU", harness.NodeSpec(arch="arm64", clusters=4, mem="L3"), parent="
       XavierSoC")
hw_graph.add_node("VisionCluster", harness.NodeSpec(DLA_Cluster=2, PVA_Cluster=2, mem="SRAM"),
       parent="XavierSoC")
hw_graph.add_node("DRAM", harness.NodeSpec(type="LPDDR4x", memSize="8GB"), parent="XavierSoC")

# Interconnect each computation node in Layer 1 to SharedLPDDR4x with AXI edges
hw_graph.add_edge("ARM_Cluster, GPU, VisionCluster, DRAM, VideoEncoder", link="AXI", bandwidth="137
       GB/s")

# Create "Reproject" TASK object
task = harness.TASK(ID="Reproject", PU=["CPU","GPU","VIC"], constraint=[harness.LatencyDeadline("30ms"
       )], inputSize, outputSize)

# Register pre-compiled binaries for the "Reproject" task
harness.registerBinary(task, "CPU", ".../bin/reproject_CPU")
harness.registerBinary(task, "GPU", ".../bin/reproject_GPU")
harness.registerBinary(task, "VIC", ".../bin/reproject_VIC")

# Implement the predict() function for PU-TASK pair.
def predict(task):
    profileData = harness.loadProfileinfo(f"/profiles/{task.ID}_{task.PU}.json")
    return {"latency": profileData.get("latency"), "resourceUsage": profileData.get("memUsage","cacheUsage")}

# Implement getComputePath() function for the GPU (implementations for other PUs are omitted)
def getComputePath(task):
    harness.Paths["Reproject"] = {
    "GPU": [XavierSoC, VoltaArch, DRAM] }
    return harness.Paths.get(task.ID)

# local ORC invokes mapTask (given in Algorithm 1) to find a target PU either locally or remotely
targetPU = orc.MapTask(task, constraint=task["Constraint"], dependency="Decode", objective="min_latency")

# Execute the task locally or remotely, as determined by targetPU
output = harness.ExecuteTask(task, input_data="/input/frames.bin", output="/output/result.bin", targetPU=
       targetPU)
```

for the `targetPU`, along with the input and output data locations. This function blocks until the execution of the task(s) finishes.

The *HARNESS* framework and the applications we use in this paper are available for download at `https://github.com/hpsslab/harness`.

5.4 Experimental Setup

In this section, we explain the two applications we tested *HARNESS* against and our system configuration.

5.4.1 Experimented Edge-cloud Applications

We demonstrate the need for *HARNESS* in two real-life scenarios from disparate domains.

Application 1 (VR) cloud-rendered VR: This application is explained in §5.2 and experimentation detail is given in §5.4.2.

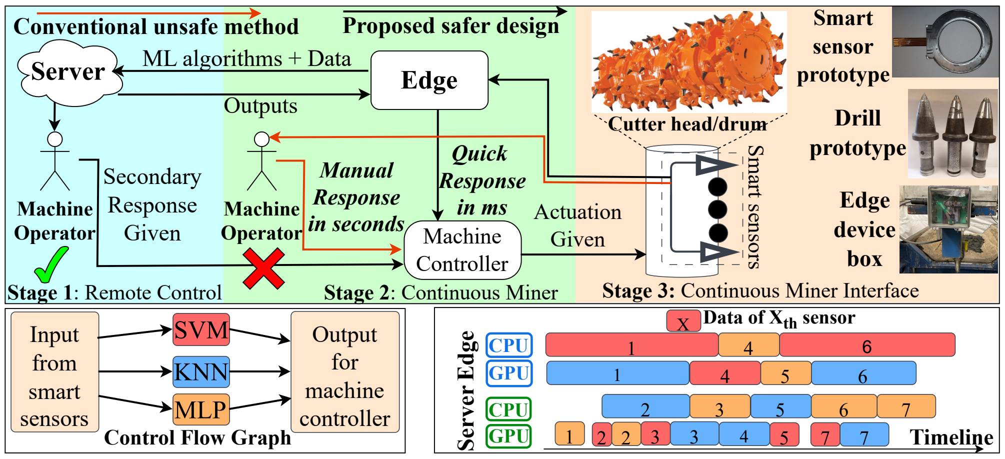


Figure 5.7 (Upper) The system operation for the Mining app. (Lower) Corresponding CFG and the pipeline of tasks.

Application 2 (Mining) smart drill bits: Underground mining requires operators to be close to the drilling machine to monitor conditions and the rock type being cut in order to prevent excessive damage to the drill bits and the mining machines. To minimize fatalities [228] and allow the operator to perform their role from a safer distance, an in-the-field, edge-based real-time data analysis system has been developed in collaboration with Mining and Electrical engineering disciplines. In this application, we read the data in real-time through multiple smart sensors that are attached to the back of the drills. The experimental setup is depicted in the upper right section of Figure 5.7. The application employs three machine learning (ML) tasks—support vector machine (SVM), k-nearest neighbor (KNN), and multi-layer perceptron

(MLP)—to process smart sensor data. These tasks can be executed in parallel as shown in the lower section of Figure 5.7. Since the cutter head drum, equipped with multiple smart sensors, rotates, the sensor data must be processed on edge devices and servers in real-time to identify the type of material being cut. If one of the ML algorithms detects an anomaly (*e.g.*, the rock type has changed), the system signals the machine controller to halt the operation.

Table 5.2 List of targeted edge devices and servers.

Edge Devices:	Orin AGX	Xavier AGX	Orin Nano	Xavier NX
Server-1:	NVIDIA Titan RTX & AMD EPYC 7402			
Server-2:	NVIDIA GeForce RTX 3080 Ti & Intel i9-11900K			
Server-3:	AMD Ryzen 5800H & AMD Graphics			

5.4.2 System Configuration

Table 5.2 lists the four edge devices with heterogeneous SoCs and the three servers with GPUs that we use in our experiments. The two diverse applications we tested *HARNESS* with are composed of numerous and different types of tasks. Figure 5.8 lists the standalone, per-task execution times for each PU in the system, for both applications and also server-edge communication time for the VR app. Tasks in VR can target up to 3 PUs, which are CPU, GPU, and VIC, whereas ML tasks in Mining can run on the CPUs and GPUs of each server and edge device. The data arrival frequency (*i.e.*, injection rate) is the FPS value for each device in VR (*e.g.*, 30 FPS at 720p for Orin AGX) and 10 Hz per sensor in Mining. We utilize PyTorch for ML tasks in the Mining application. In edge devices, we use JetPack [229] 5.1.1 version and VPI [230]. Edges and servers are connected through WAN having a capacity of 10 Gb/s per device. Our slowdown model builds upon PCCS [93]. The novelty of our approach comes from (i) scalable integration of PCCS into the multi-tiered hierarchical *ORC* mechanism, and (ii) the decoupled approach that eliminates the need for pair-wise profiling of each task. On server GPUs, we estimate multi-tenancy-caused slowdown via profiling and empirical methodologies proposed by [88, 89]. We profile standalone execution times and slowdown characterization only once for each task/PU pair. Since the *HARNESS* enables decoupling the slowdown calculation, there is no need for pair-wise execution of potentially collocated tasks.

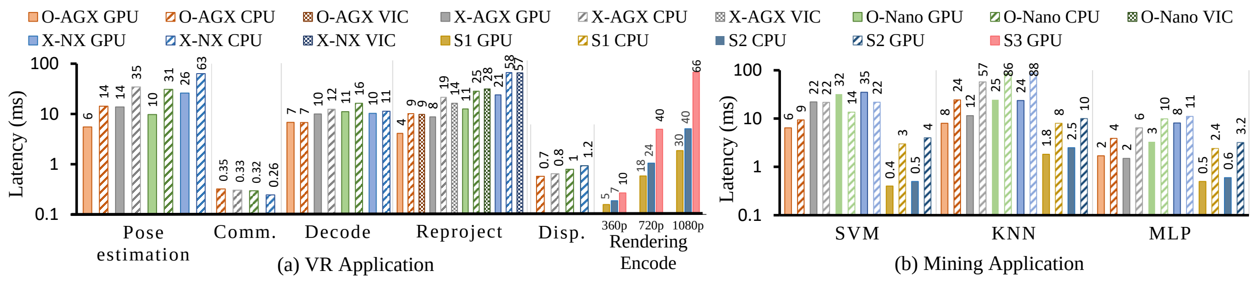


Figure 5.8 Standalone execution times of the tasks in VR and Mining apps for the various PUs of the edge devices and servers.

Baselines: We compare against three studies:

LaTS [22] proposes a latency-aware task scheduling algorithm for real-time vision applications on heterogeneous edge-cloud systems. LaTS benchmarks the performance of system per task, periodically monitors the availability of PUs, and dynamically assigns the tasks based on the standalone execution time on PUs. However, LaTS does not utilize a shared resource contention mechanism.

ACE [21] constructs a unified platform for edge-cloud platforms considering high scalability. Yet, ACE is limited to static mappings only. So, ACE struggles to adapt to the changes throughout the execution and does not consider shared resource slowdown within a node.

Multi-tier CloudVR [221] specifically addresses the challenges associated with real-time remote VR rendering. Cloud-VR is adaptable to dynamically changing network conditions by balancing the computation and communication time by shrinking the frame resolution. However, CloudVR does not handle tasks other than rendering in the VR app.

The two baselines we compare against (*i.e.*, *ACE* [21] and *LaTS* [22]) are not shared-resource aware and ACE is also not capable of adapting to dynamic changes. So, the experiments in §5.5.1 and §5.5.2 comparing *HARNESS* against these baselines serve as an implicit ablation study for *HARNESS*.

5.5 Evaluation

In this section, we assess the utility of *HARNESS* with various experiments. For the first two experiments given in §5.5.1 and §5.5.2, we use VR and Mining apps, respectively. For other experiments, we interchangeably use both applications.

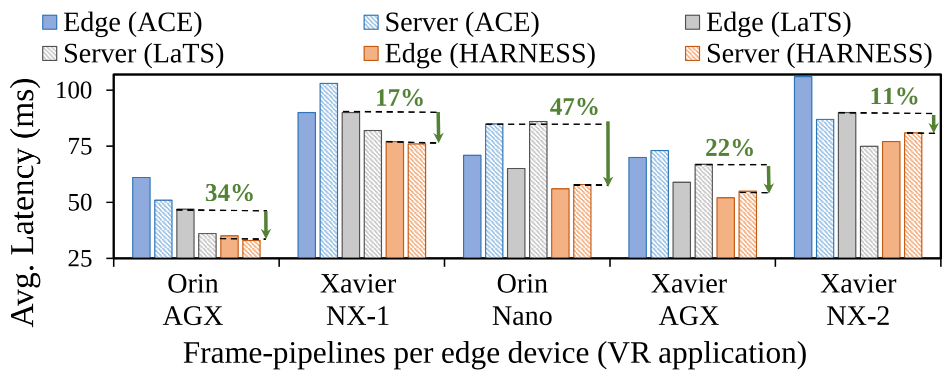


Figure 5.9 Average overall latency of *HARNESS* and others.

5.5.1 Overall Performance

In this main experiment, we assess *HARNESS*'s performance using a DECS comprising five edge devices and three servers (one from each with two Xavier NX). We define *latency* as the total time of a single VR frame spent on edge and server. Our goal is to minimize the pipelined latency of a frame in the VR app. Our results are reported in Figure 5.9. *Overall, HARNESS improves pipeline latency from 11% to 47% over the best baseline.* During the experiments, we observe that LaTS and ACE prioritize assigning *reproject* task to edge's CPU (if available) on Orin AGX, Xavier AGX, and Xavier NX over VIC since CPU's standalone time is superior over VIC. Yet, under the shared memory contention by multiple PUs, CPU generally performs worse than VIC since CPU shares the L4 cache with GPU and VIC has private buffers optimized for such tasks to minimize memory accesses. Additionally, LaTS and ACE choose to perform pose estimation on the edge devices' GPU and CPU, resulting in underutilized server resources and oversubscription of edge resources.

The average per-frame latency difference between all edge-server pairs is 11.8% for ACE, 12.6% for LaTS, and 2.4% for *HARNESS*, highlighting *HARNESS*'s adeptness at offering balanced system resource utilization. Among the pipelines of the five edge devices given in Figure 5.9, the bottlenecks are on the server-side for the last three and on the edge-side for the first two. Given that servers are the bottleneck in three instances, we deduce that adding an extra server could enhance the performance of overall system.

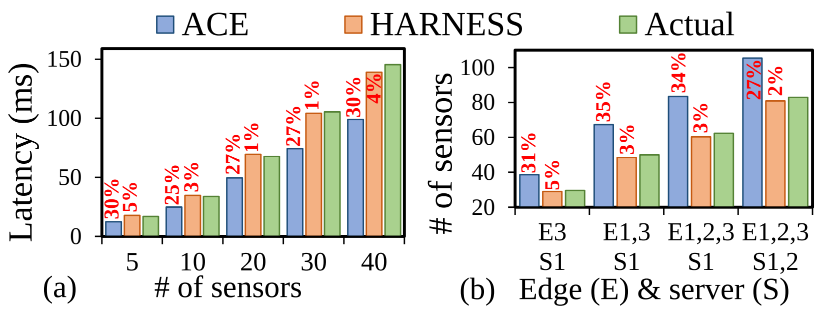


Figure 5.10 Prediction error rates of ACE & *HARNESS* for a system with (a) 1 edge device + server, (b) increased # devices.

5.5.2 Model Validation

We validate the performance prediction accuracy of *HARNESS* in the Mining app and compare it to a baseline model (ACE). In the first experiment, the objective is to determine the maximum number of smart sensor readings that both an edge device and a server (*e.g.*, Orin Nano and server 1 in this experiment) could process within a 100 ms latency threshold. We define *latency* in the Mining app as the time passed from the data being read by the sensor until all three ML tasks (*i.e.*, SVM, KNN, and MLP) are completed. This latency includes computation, slowdown, communication time, and the overhead spent to schedule the task. Figure 5.10.a depicts the performance predictions made by ACE and *HARNESS* and compares them to the actual time it took to run. *Notably, HARNESS shows an average prediction error rate of 3.2%, significantly lower than 27.4% error rate of ACE.* A critical insight emerges for the experiments involving 30 and 40 sensors: ACE inaccurately predicts that tasks could be completed under the 100 ms latency threshold while *HARNESS* considers other factors, such as shared resource interference.

In the second experiment depicted in Figure 5.10.b, we gradually increment the number of edge devices (Orin AGX-E1, Xavier AGX-E2, and Orin Nano-E3) and servers (server 1 and 2) in the system. Our goal is to determine an upper bound on the number of sensors that the DECS HW could handle under 100 ms latency. *HARNESS* can predict this with up to 98% accuracy whereas ACE overlooks the contention-related slowdowns and overloads slower edge devices, resulting in a falsely optimistic sensor count estimation.

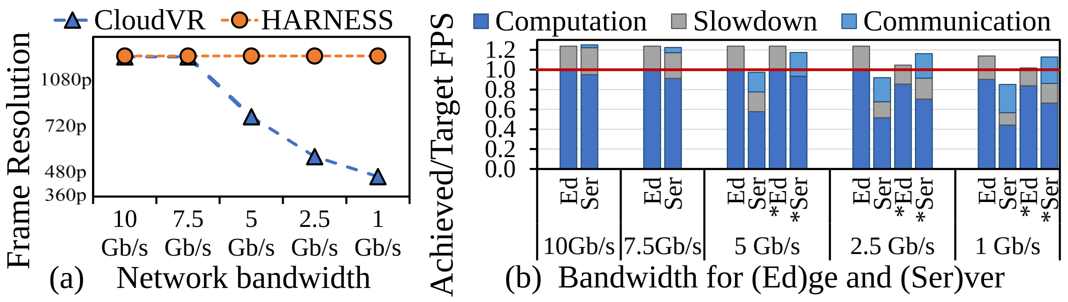


Figure 5.11 The variations in (a) video quality and (b) targeted FPS for changing network conditions. The Ed(ge) and Ser(ver) bars without * and with * denote the breakdown before and after *HARNESS* adapts to the change, respectively.

5.5.3 Dynamic Adaptability

When network conditions degrade for an edge device, Multi-tier CloudVR [221] proposes decreasing the frame resolution at runtime to keep up the target FPS by reducing both computation (*e.g.*, some listed in Figure 5.8) and communication time, whereas *HARNESS* can hierarchically update the scheduling assignments (*i.e.*, dynamically adapt) while also considering the delay introduced by the network communication since the *ORC* mechanism is triggered for every task assignment. We gradually decrease the network bandwidth capacity from 10 Gb/s to 1 Gb/s and analyze the change needed for the frame resolution to meet a fixed FPS target with the same experiment design in §5.5.1. As demonstrated in Figure 5.11.a, we observe that CloudVR targets lower frame resolutions after 7.5 Gb/s whereas *HARNESS* can keep up with FPS requirements by balancing the workloads through the entire system.

To gain a deeper insight into *HARNESS*'s ability to handle dynamic workload assignments, we further analyze the time breakdown of computation, slowdown, and communication on Orin AGX and target server(s). Figure 5.11.b presents the ratio of the average achieved FPS over the targeted FPS. When bandwidth is reduced to 7.5 Gb/s, *HARNESS* successfully maintains the target FPS above the predefined threshold while still running mostly on server-2 as in the 10 Gb/s scenario. At a reduced bandwidth of 5 Gb/s, the *ORC* continues to assign the rendering task to server-2 for Orin AGX, but it avoids server-side GPU sharing. This alleviates the additional communication overhead on the server-side and maintains the target FPS for every edge device in the system since the rendering task on other edges can be assigned to different servers. In the most constrained scenario, at 1 Gb/s, *HARNESS* proactively identifies and assigns the rendering task to the best matching server (server-1) that can meet the target FPS requirements. Meanwhile, rendering tasks previously running on server-1 are reassigned to server-2.

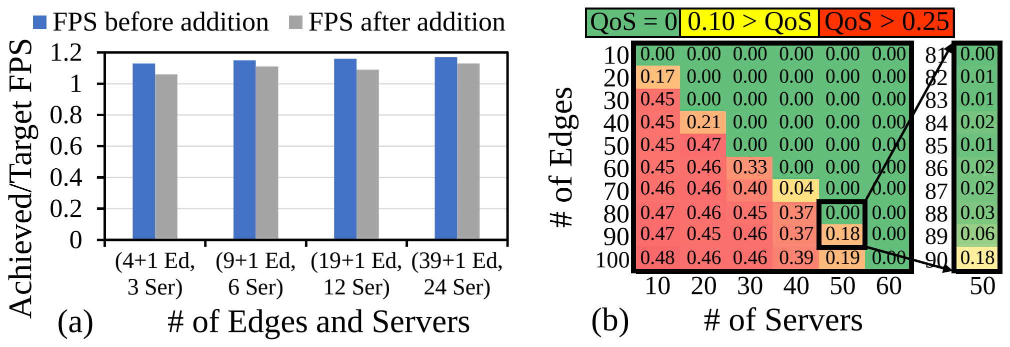


Figure 5.12 (a) *HARNESS* adaptability during edge device additions. (b) QoS violations as number of nodes scales.

When a new edge device joins an active edge-server system, a server must be assigned or shared to handle, at a minimum, the rendering and encoding tasks for the new device. This necessitates the recalculation and rescheduling of multiple rendering tasks across servers. After the edge-device is connected, we dynamically add the device to our *HW-GRAPH* and the next time the *ORC* is called to map a task, it considers the new change. Figure 5.12.a depicts how *HARNESS* maintains the desired FPS for varying server-edge counts. The blue bar denotes the worst FPS among edge-system pairs before the new device and the gray bar shows the FPS after *HARNESS* handles the workload changes.

To analyze the QoS failure for varying edge/server ratios, we gradually increase the number of edges and servers by 10 and measure QoS failure per frame (Figure 5.12.b). We report the average QoS failure as the total number of frames violating the latency requirement over the frames completed. We observe that DECSs having more than or equal to a 2-to-1 edge/server ratio will result in noticeably high rate of failures since individual servers struggle to process more data than two edge devices provide.

While the experiments so far are the results of real-world testing, the experiments given in Figure 5.12 and the subsequent experiments in the §5.5.4 rely on simulations that use the individual edge/server profiles validated in §5.5.2.

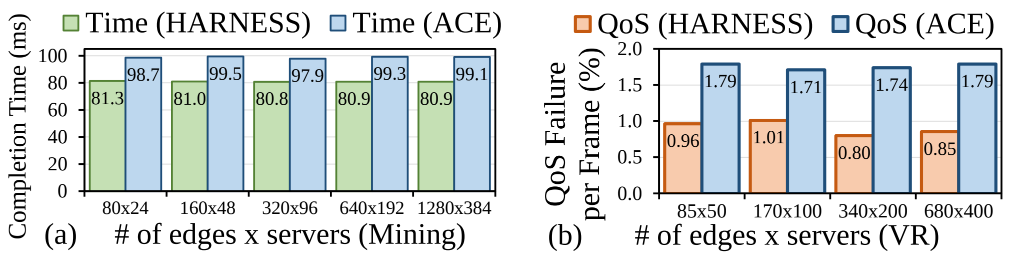


Figure 5.13 Weak scaling experiments for (a)Mining and (b)VR.

5.5.4 Scalability

Scaling experiments design: Weak scaling evaluates *HARNESS*'s performance while increasing the number of computing devices and keeping the average number of tasks per computing device constant. Strong scaling keeps the total number of tasks in the system fixed while the number of edge devices and servers is proportionally increased. In the Mining app, we measure the total completion time of the tasks which includes computation, data transfer, and communication between devices, slowdown per PU in each device, and scheduling overhead of *HARNESS* per task. In the VR app, as edge devices increase, the number of tasks also grows, as each device adds more frames and, consequently, more workload to be processed. The times reported for the VR app are comprised of similar contributors as with the Mining app. We compare the scalability of *HARNESS* against ACE [21], which we identify as the most relevant state-of-the-art study.

Weak scaling-1: For the Mining app, our initial setup includes 100 smart sensors with 80 edge devices and 24 servers (20 and 8 of each edge device and server listed in Table 5.2, respectively). Each experiment doubles the number of smart sensors, edge devices, and servers in the previous setup. Figure 5.13.a reports the completion time per setup. *HARNESS* keeps the trend of completion time around 81ms as the number of devices and input sources proportionally increases whereas ACE generally keeps the completion time around 98ms due to the under-utilization of the system.

Weak scaling-2: For the VR app, we start with 85 edge devices with 50 servers and double them for each setup. Results shown in Figure 5.13.b demonstrate that, for *HARNESS*, QoS failure rate is kept minimal as the system scales. Even though ACE has a similar trend in both applications, lack of contention handling and static assignment of tasks leads to under-utilization of the system, resulting in longer completion times and higher QoS failures.

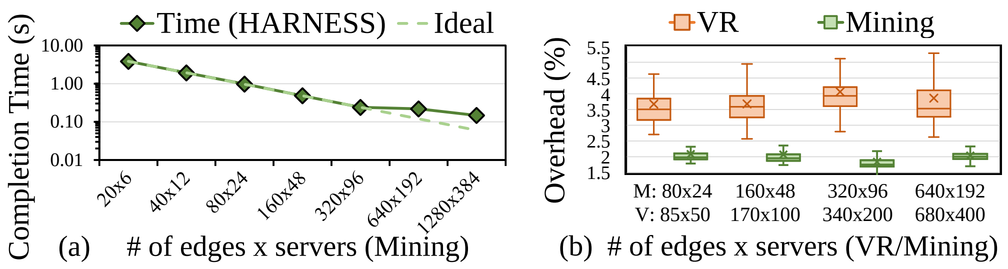


Figure 5.14 (a) Strong scaling experiments, (b) *ORC* overhead.

Strong scaling: In the Mining app, we deploy concurrent 1250 sensors each of which triggers three ML tasks. Results reported in Figure 5.14.a show a linear decrease in completion time up to the configuration of 640 edge devices. Beyond this point, the performance is principally constrained by the KNN task execution time on the Xavier NX edge devices, which emerge as the primary bottleneck.

5.5.5 *ORC* Overhead

We define the *ORC* overhead for a given task as the ratio of the initial `MapTask()` latency over the execution latency of the task. This time includes the computation time by the traverse and the time spent for communication and computation between local and remote *ORC*s until a PU is found.

DECS size: As the number of edges and servers are doubled, we measure the *ORC* overhead per task, average them to calculate the total overhead per iteration of both applications, and report the distribution in Figure 5.14.b. The scheduling overhead is consistently preserved around 2% for Mining and 4% for VR apps. We observe that more than 90% of the overhead originates from communication between ORCs located on different edge devices or servers. If ORC is local, there is no communication and the overhead of running *Traverser* on the local *HW-GRAPH* is minimal (around 0.1ms on average). If an ORC needs to communicate with remote ORCs over the network, every hop adds around 0.3ms, with our experiments having 2 hops max. Each ORC communication transmits a minimal amount of information over the network (*i.e.*, the parameters of the `MapTask()` function, as shown in §5.3.6). It is important to note that profiling times are not included in the overhead calculations since profiling is performed only once per task-PU pair.

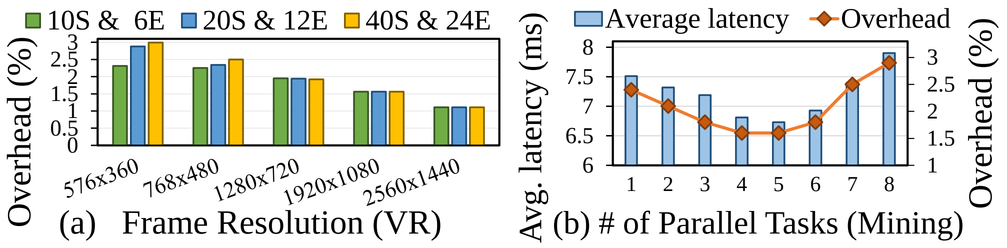


Figure 5.15 (a) Effect of task size over overhead for [E]dge & [S]erver pairs. (b) Granularity of parallel tasks assigned.

Task sizes: We investigate the effect of task sizes (*e.g.*, frame resolution size in the VR app) over the overhead of *HARNESS* by varying frame resolution while reducing the latency (*i.e.*, FPS) requirements per device proportionally. The results reported in Figure 5.15.a show that, as the input size gets bigger, the number of tasks sent to servers per second lowers, thus decreasing the overhead of *HARNESS* per frame.

Mapping granularity: We investigate the effects of calling `MapTask()` with an increasing number of tasks and observe its relationship to the *ORC* overhead. Figure 5.15.b shows the results for the Mining app. We observe that the average latency and overhead decreases until we assign 5 parallel tasks. However, mapping larger batches of tasks leads to load imbalance between computational nodes (especially in edge devices) because `MapTask()` is designed to assign a given set of tasks only to the PUs in the same computational node.

5.5.6 Handling Prediction Errors

We use empirical profiling in our evaluations; therefore, our prediction error rates are minimal, as shown in §5.5.2. *Traverser* embeds a window-based correction mechanism to address larger and continuous deviations in performance predictions (see §5.3.4 for details). To demonstrate the effectiveness of this correction mechanism, we conduct an experiment in which we artificially introduce delays in task executions to emulate performance mispredictions.

In this experiment, whose results are shown in **??**, we run the Mining application continuously with five different sensor counts. We vary the artificial delay added in task executions between three intervals: 5%-10%, 10%-25% and 25%-50% of the original execution times. We report the resulting slowdown in the average task execution time compared to an Oracle that has perfect knowledge of mispredicted execution times. The graphs on the left and right side of **??** show how *HARNESS* performs without and with the misprediction correction mechanism, respectively. Our experiments show that the proposed correction mechanism in *HARNESS* is capable of mitigating the side effects of mispredictions to a considerable extent. As the added delay is higher, the resulting slowdown is less (compared to the delay to execution time ratio), due to the averaging of the predictions over a running window. In addition, we also observe that the negative effects of misprediction are less significant when the sensor count is increased. This is due to the increased amount of opportunities for overlapping present with higher sensors count.

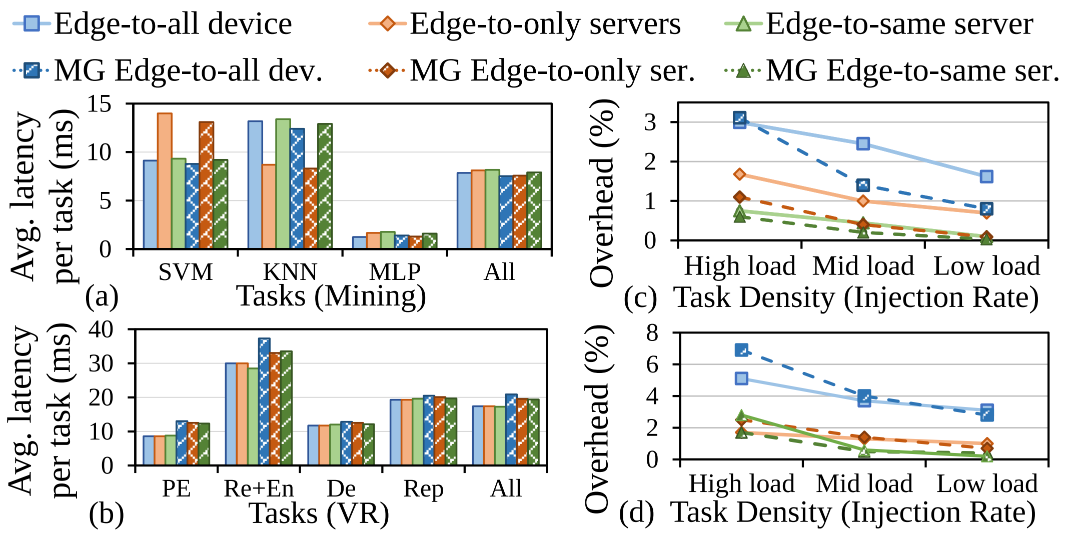


Figure 5.16 Latency and overhead for assignment strategies.

5.5.7 Various Mapping Strategies

In the default mapping policy of *HARNESS*, as given in Alg. 1, local *ORC* checks child and parent *ORC*s hierarchically. Here, we explore alternate strategies. The first one (*i.e.*, Edge-to-only servers) involves direct communication from edge devices to servers, bypassing the communication between the *ORC*s of sibling edge devices. The second strategy (*i.e.*, Edge-to-same server) re-attempts to assign the task to the same node and PU that processed the task in the previous iteration. Lastly, we repeat the two strategies above by increasing the mapping granularity (MG) of ready tasks, similar to the previous experiment. Figure 5.16.a and Figure 5.16.b show average task latency for each strategy. In the VR app, we observe that first and second strategies improve system latency. Since the rendering task is often mapped to servers, these two strategies decrease the *ORC* overhead by skipping less powerful edge devices. In the Mining app, however, failing to query other edge devices leads to under-utilization of sibling edge devices and increases the latency. MG in the Mining app can improve the average latency whereas it does not in the VR app.

Figure 5.16.c and Figure 5.16d depict the *ORC* overhead when tasks are created in varying intervals for Mining (20 Hz, 10 Hz, and 5 Hz) and VR (1.10x, 1x, and 0.75x FPS of default values). We observe that frequent task creation has a higher overhead since the *ORC*s communicate more. In the VR app, grouping tasks generally causes higher overhead because some tasks cannot be mapped to a PU under current latency constraints and we end up splitting the tasks and rescheduling. This pattern is also observed in the Mining app under high load. Reprojection task is the only task that can run on VIC, yet during high load experiments, VIC may not successfully complete the task under stricter QoS requirements due to shared resource slowdown, which other baselines cannot spot. This results in using GPUs instead and leaving VIC idle. Please note that splitting is applied only when the user specifies a group of TASKs at once for the MapTask() API call. Single `TASKs` are not split into smaller pieces.

5.6 Conclusion

We propose *HARNESS*, a holistic resource management framework tailored for diversely heterogeneous edge-cloud systems. *HARNESS* uniquely takes the slowdown due to shared resource usage into account. We demonstrate the utility of *HARNESS* on two real-life applications from disparate disciplines deployed on the field, yielding 47% reduction in latency over baselines with less than 2% scheduling overhead.

CHAPTER 6
MEMORY CONTENTION-BASED COVERT CHANNEL COMMUNICATION ON SHARED DRAM SYSTEM-ON-CHIPS

After observing the contention amounts over shared memory, as in Chapters 4 and 5, we explored how contention behavior can be exploited as a covert channel attack. In this chapter, we will introduce MC^3, a novel high-throughput covert channel attack over shared DRAM.

6.1 Introduction

Mobile system-on-chips (SoCs) house multiple types of processing units (PUs), including general-purpose CPU cores and domain-specific accelerators (DSAs), such as GPUs and deep learning accelerators. With the proliferation of integrated DSAs, modern SoCs can provide cost-effective and energy-efficient execution, making them ideal candidates for in-the-field computing in many areas (mobile phones [231], smart home environments [232] and autonomous systems [196]).

An emerging architectural feature of modern SoCs (*e.g.*, NVIDIA's Orin [196], Apple's M3 [233], Qualcomm's Snapdragon [234]) is a shared main memory where the data is stored for access by all PUs. The use of shared physical memory (SM) in commodity SoCs is motivated by the goal of reducing the chip area and production costs. It can also provide additional performance benefits by minimizing data transfer overhead between the CPU and the DSAs [235]. However, several studies [39, 93, 94] revealed that when running multiple workloads concurrently, PUs in SM-SoCs can experience significant slowdown caused by shared memory contention. Due to the diverse computational characteristics of the PUs they embed, SM-SoCs often do not employ a shared last-level cache (LLC). Although covert channel attacks have been widely studied in shared memory systems, high-throughput communication has previously been feasible only by relying on an LLC or by possessing privileged or physical access to the shared memory subsystem.

Constructing a fine-grained, low-noise, and high-throughput memory-contention-based covert channel attack on mobile SM-SoCs presents several challenges: (i) SM-SoCs generally lack a shared LLC to avoid complex design requirements. The trojan (*i.e.*, transmitter) needs to generate sufficient memory pressure that is observable by the spy (*i.e.*, receiver). CPU-based workloads in resource-limited SM-SoCs often fail to fully utilize the memory bandwidth even when all cores are used [93]. Therefore, accelerators with higher memory demands, such as GPUs, should be employed. Meanwhile, the generated memory pressure should be low enough to minimize the risk of being detected by system defenses. (ii) The total memory pressure exerted cumulatively by the spy and the trojan must be reliably high. Memory accesses satisfied by the caches can artificially increase perceived bandwidth, hence, they should be minimized to achieve a reliable access stream reaching the DRAM. This is also crucial to maximize the capacity of the communication channel. (iii) Without external synchronization mechanisms, reliable and high-throughput data transmission over SM becomes challenging, as the trojan and the spy may operate at different magnitudes of memory operations, particularly when located on different types of PUs (*e.g.*, CPU cores and GPU). (iv) Finally, these requirements should be achieved without elevated privileges and hardware access, and the attack should function under single-user and multi-application environments. Our proposed work addresses all four challenges listed above.

In this paper, we introduce a new memory contention-based covert communication attack, MC^3, targeting shared memory SoCs on mobile platforms. Our attack exploits the underlying vulnerability with software-only mechanisms, requiring neither direct hardware access nor super-user privileges. MC^3 is designed to achieve a balance between the communication accuracy and the transmission rate (*i.e.*, capacity) of the covert channel. To increase the transmission rate and improve the efficiency of our attack, we further propose a CPU+GPU version of the receiver and the transmitter. We demonstrate that MC^3 achieves transmission rates of up to 6.4 Kbps with an error rate below 1% in CPU-to-GPU communication. Our implementation is available at `https://github.com/hypesys/MC3`.

6.1.1 Contributions

Our work makes the following contributions:

- We unveil a new attack vector that leverages the slowdown in memory accesses due to shared use of system memory. The attack vector is achieved through software-only measurements and does not require privileged access to the system.
- We present a novel covert channel attack that targets shared-memory SoCs without a last-level cache between its processing units. Our attack considers both CPU-GPU and CPU-CPU placements of the transmitter and the receiver.
- We evaluate MC^3 on NVIDIA Orin AGX, Nano, and NX SoCs and achieve a channel capacity of up to 6.4 Kbps with 95% accuracy. The accuracy reaches 99.99% when the capacity is capped at 1.3 Kpbs.

The remainder of this chapter is organized as follows: Section 6.2 reviews the background and motivation behind shared-memory covert channels. Section 6.3 presents the MC^3 attack in detail, covering the threat model, mechanism design, contention primitives, transmitter/receiver architecture, and noise-reduction techniques. Section 6.4 explores enhancements using GPUs and analyzes channel capacity, accuracy, and trade-offs.

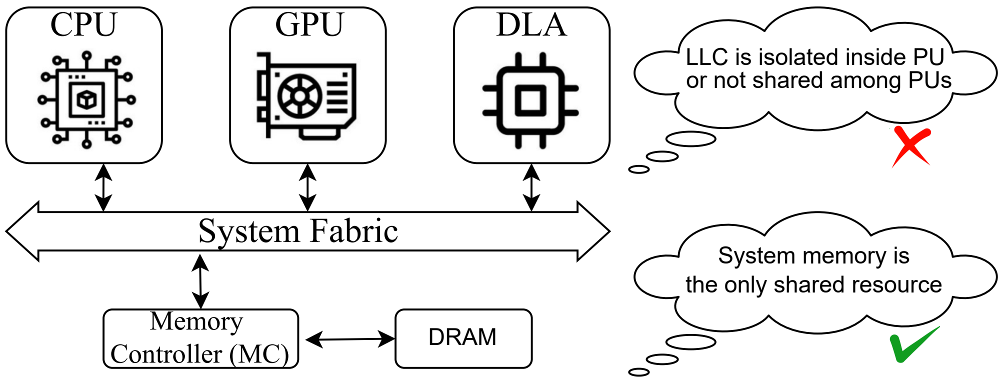

CPU
GPU
DLA
LLC is isolated inside PU or not shared among PUs
System Fabric
System memory is the only shared resource
Memory Controller (MC)
DRAM

Figure 6.1 Block diagram for NVIDIA's Xavier AGX SoC embedding a CPU, GPU, deep learning accelerator (DLA) and shared memory.

6.2 Background and Challenges

Shared memory SoCs (SM-SoC): Modern SoCs, such as NVIDIA's Xavier and Orin architectures (as depicted in Figure 6.1) integrate different types of accelerators, such as GPUs and DSAs, and each is optimized for specific computations. Unlike larger-scale systems where each accelerator has a dedicated primary memory, SM-SoCs share a common DRAM-based memory such as DDR4. Each PU has access to memory via a shared memory bus and a centralized memory controller (MC). Due to their inherent architectural heterogeneity, SM-SoCs often lack a shared LLC. Modern SoCs benefit from shared memory design because it reduces production costs and improves the data transfer overhead between CPU, GPU and other PUs.

Shared memory contention as attack vector: Our proposed methodology relies on the vulnerability that we discover in shared-memory SoCs and has the potential to be exploited on mobile and autonomous SoCs for a variety of attacks that do not require privileged access. A programmer can develop an adversarial transmitter application that leaves a distinct signature via purposeful shared memory accesses. This signature can be used to leak crucial information that can be encoded in binary form.

Although this attack strategy seems similar to other types of covert channel attacks, there are unique challenges to efficiently and reliably designing shared memory contention channels:

- *Sufficiently observable contention:* While fully stressing memory resources to maximize contention is technically possible by the use of accelerators, the attack should generate sufficiently enough contention for the transmitter and receiver to communicate with each other. This will allow the attack to remain undetected by countermeasures deployed by the OS while maximizing the channel capacity of the attack. We deal with this challenge by creating contention that only targets shared memory resources and bypasses the private cache hierarchy of CPUs.

- *Reliable and efficient contention:* Achieving reliable contention generation necessitates a careful characterization of the channel's behavior. While reliability can be increased by repeatedly performing contention for a long time to transmit a bit, the practicality of the attack often requires minimizing the repetition. We overcome these challenges by thoroughly analyzing the contention behavior and using fine-grained time intervals for the receiver and the transmitter.
- *Synchronizing transmitter and receiver:* Unlike traditional cache attacks, where the effects of a cache hit-or-miss can be clearly observed in the order of nanoseconds, the slowdown caused by the memory contention becomes visible in the order of microseconds. This requires synchronized transmitter and receiver operation. Additionally, considering the differences in computational capabilities and clock rates of the CPUs and GPUs, the design of attack vectors on two diverse PUs requires the synchronization to be done without using any external resources. We overcome this challenge by developing a precise contention generator and sleep procedure for the transmitter and adapting them to the receiver accordingly.

Feasibility of shared memory contention-based covert channel: To demonstrate how the memory contention behavior affects the observed memory bandwidth of an application, we run the transmitter app on the CPU and the receiver on both the CPU and the GPU of an Orin NX SoC. Figure 6.2 shows two raw traces of the varying average bandwidth (BW) perceived on the receiver side. Regardless of whether the receiver runs on the CPU or the GPU, the perceived BWs for the receiver have clear drops in the traces, which correspond to the '1's sent by the transmitter. This experiment demonstrates the feasibility of building covert channels with shared memory contention.

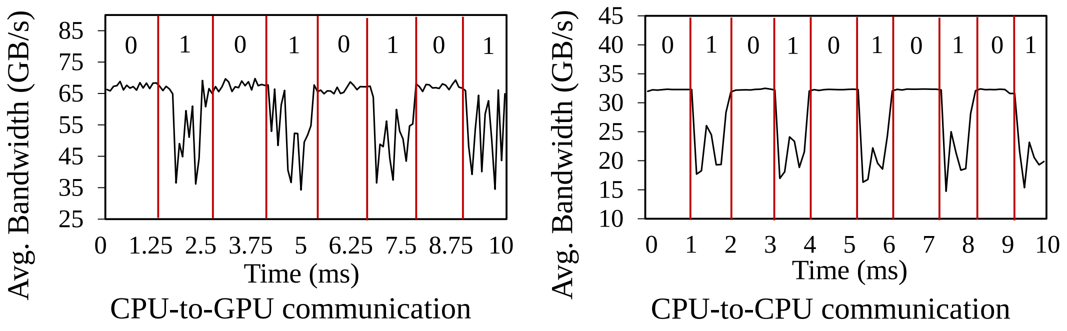


Figure 6.2 Raw traces for CPU-to-GPU and CPU-to-CPU communication

6.3 Shared Memory Contention-based Covert Channel Communication

In this subsection, we present the threat model, MC^3 methodology, attack vector details, the precision contention duration and sleep, and the trade-off between transmitter and receiver designs.

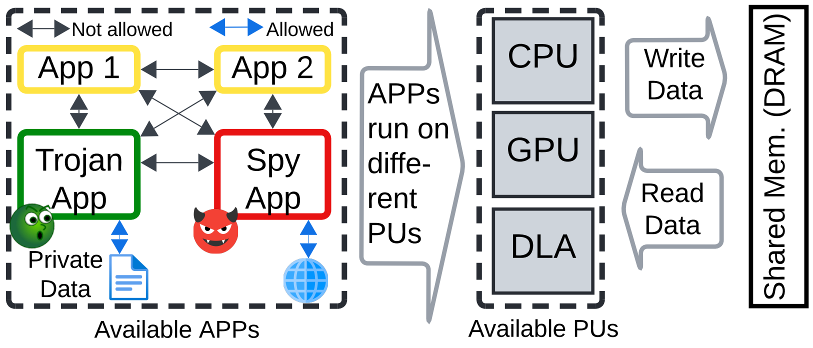


Figure 6.3 Threat Model

6.3.1 Threat Model

Figure 6.3 depicts our threat model which involves running two (or more) applications on an SM-SoC (as explained in Section 6.2). The transmitter (*i.e.*, trojan) is an application that has access to sensitive or private user data. The receiver (*i.e.*, spy) is an application running on the same SM-SoC but does not have access to the same data. Applications running in the system (including the receiver and transmitter) are not allowed to communicate with each other. Both transmitter and receiver can run on the CPU or GPU (in no specific order) without elevated execution privileges —meaning they cannot access protected OS facilities or performance counters. The attack is designed to be executed remotely and attacker's presence or active engagement is not required. The attacker is assumed to have no physical access to the hardware components (*e.g.*, for measuring power consumption and electromagnetic emissions).

6.3.2 Overall Mechanism

Figure 6.4 illustrates the communication protocol between the transmitter and receiver for transmitting bits (*i.e.*, 0 or 1) through shared memory contention. The transmitter conveys bits by performing buffer copy operations on memory while the receiver continuously performs another buffer copy operation to detect the transmitted bit.

The transmitter is responsible for sending the bit by modulating the level of memory contention. As shown in the upper part of Figure 6.4, to transmit a bit '0', the transmitter sleeps for a predefined time interval of T_n. To send a bit '1', it performs continuous copy operations to access DRAM. The receiver continuously operates its own buffer copy function, then measures the duration of the buffer copy operation, and finally calculates the average BW over the duration, which will be used to decode the bit.

The lower part of Figure 6.4 illustrates the case where the receiver can perform more copy operations with lower latency (*i.e.*, higher memory BW) while the transmitter is sleeping. Multiple copy operations per time interval T_n can also be utilized to have more reliable data transmission (see Sec. 6.3.8). In contrast, when the transmitter induces contention by performing a copy operation, the receiver's throughput decreases (*i.e.*, lower memory BW) because memory latency increases due to shared memory contention.

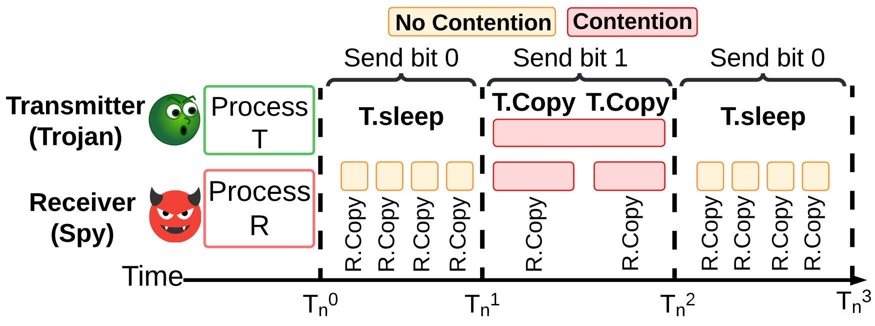


Figure 6.4 Communication between the transmitter and receiver.

While it is possible to run the receiver non-stop, we instead opt to start the receiver slightly earlier (*e.g.*, one second) than the transmitter at a predetermined epoch. This design decision eliminates the need for continuous operation of the receiver, thus minimizing the chances of our attack being detected under real-world conditions. The early start allows the receiver to collect BW information (*i.e.*, latency per copy operation) without any interference from the transmitter, which can be used as a baseline during the data analysis stage. Although other applications on the device may use shared memory —potentially introducing noise into the receiver's average BW measurements— the pattern of zeros and ones can be detected using heuristic history-based signal processing approaches.

Table 6.1 lists our test devices with varying computational capability and memory BW capacity. We use Jetpack 5.1 on all devices. It is worth noting that all devices have TrustZone Trusted Execution Environment (TEE) and OS-protection regions in the memory subsystem.

Table 6.1 Targeted platforms

Device	Orin AGX	Orin Nano	Orin NX
CPU	12-core A78	6-core A78AE	8-core A78AE
GPU	2048 core Ampere	1024 core Ampere	1024 core Ampere
DRAM Size	64 GB 256-bit	8 GB 128-bit	8 GB 128-bit
Memory Bandwidth	204.8 GB/s	68 GB/s	102 GB/s

6.3.3 Attack Vector

Our attack leverages a memory-contention channel to stealthily transmit data between two processes.

Our transmitter algorithm, described in Alg. 2, is responsible for encoding the data and transmitting the encoded data bitstream over the memory-contention channel. It loops through the input data bitstream, either running the memory contention kernel (*i.e.*, data copy operation) for the specified duration T if the current bit is a 1, resulting in memory contention, or sleeping for the same duration T (if not set differently) if the current bit is a 0, resulting in near zero memory contention.

Our receiver algorithm, described in Alg. 3, is responsible for receiving the encoded data from the memory-contention channel and decoding it. It continuously runs the memory contention kernel over the duration T —the T value should be the same as the transmitter's. Then, it normalizes the BWs based on a global average (*i.e.*, subtract each BW sample from the overall average observed BW) [126]. Then, this normalized BW is thresholded with hysteresis, which we experimentally determined to be much more resistant to noise. Finally, the result of this thresholding is converted directly into the received bitstream, where the values above the threshold become 0 (which corresponds to a higher measured BW in the receiver, as a result of the transmitter *not* simultaneously generating memory contention) and the values below the additive inverse (*i.e.*, $-1\times$) of the threshold are 1 (which corresponds to a lower measured BW in the receiver, as a result of the transmitter simultaneously generating memory contention).

Algorithm 2 Transmitter

```
Input: data bitstream $B$ and its length $n$, time interval $T$
  for $i \leftarrow 0$ to $n-1$ do
      if $B[i]$ is 1 then CONTEND_FOR($T$)        ▷ Generate contention
      else SLEEP_FOR($T$)                          ▷ Remain idle
```

Algorithm 3 Receiver

```
Input: hysteresis threshold $\gamma$, run length $n$, time interval $T$
  $B \leftarrow []$                                  ▷ Output bitstream (starts empty)
  $b \leftarrow 0$                                   ▷ Hysteresis state
  $\bar{\beta} \leftarrow 0$                         ▷ Average BW
  for $i \leftarrow 0$ to $n-1$ do
      $\beta_{\text{raw}} \leftarrow$ CONTEND_FOR($T$)
      $\bar{\beta} = \frac{\bar{\beta} \cdot i + \beta_{\text{raw}}}{i+1}$        ▷ Use simple global average
      $\beta_{\text{normalized}} = \beta_{\text{raw}} - \bar{\beta}$
      if $\beta_{\text{normalized}} > \gamma$ then APPEND($B$, 0)
          $b \leftarrow 1$
      else if $\beta_{\text{normalized}} < (-1 \cdot \gamma)$ then APPEND($B$, 1)
          $b \leftarrow 0$
      else APPEND($B$, $b$)
  return $B$
```

Table 6.2 Precise ‘contention’ and ‘sleep for’ durations

Operation	Expected duration	Mean Error	Minimum Error	Maximum Error	Std.;dev. Error
Sleep for	100 ms	46 ns	5 ns	314 ns	41 ns
Contention	100 ms	12 μs	5 ns	767 μs	65 μs

6.3.4 Cache-less Memory Access

Throughout the development of the memory-contention kernel, we observed that CPU caches are used to access the data and artificially inflate the memory BW measurements by using data-streaming kernels. Although sufficient contention generation using data streaming kernels is also possible, it makes our BW measurements unreliable and introduces substantial noise into the communication channel.

To alleviate this, our implementation employs memory instructions with non-temporal hints, specifically `ldnp`/`stnp` (Load/Store Pair Non-Temporal) Arm64 instructions. This hint signifies that the data being loaded or stored is unlikely to be reused soon, prompting the system to bypass the cache hierarchy [236]. By doing so, we ensure that our memory operations access DRAM directly, enhancing the reliability of our BW measurements and increasing the efficiency of the attack. It is worth noting that we also observed sufficient contention generation with data streaming using regular data streaming without non-temporal instructions.

6.3.5 Precise Contention Duration and Precise Sleep

In order to maximize performance, the sleep and data copy operations require high temporal precision (*i.e.*, sleep or run for the desired duration as accurately as possible). We implemented a precise sleep mechanism by utilizing an OS-provided function (*i.e.*, `std::this_thread::sleep_for`) until near the desired end time and finally spinning (*i.e.*, `while` loop with an empty body) until the desired end time is reached [237]. To implement `CONTEND_FOR(T)`, used in both algorithms, our data copy operation first runs the memory-contention kernel for a small, fixed amount of data (*i.e.*, taking nearly one second) to estimate the currently achievable memory contention BW (β_0). Using the total desired duration (T), it estimates the amount of data the kernel needs to run for (d_*, where $d_* \propto \beta_i \cdot T$). Using this estimate, it splits this total data estimate up (e.g.: $d_i = \frac{1}{100} \cdot d_*$), runs the memory-contention kernel (for d_i amount of data), collects β_i, updates d_* and d_i, and repeat. When it gets close to the desired end time, it further splits the estimate (e.g.: $d_i = \frac{1}{1000} \cdot d_*$) to reduce the amount of under/overshoot. In Table 6.2, we report the results for 100 ms execution, demonstrating our average error rate of 4 and 7 orders of magnitude lower for sleep and contention generation, respectively.

6.3.6 Transmitter and Receiver Design

Our attack hinges on the transmitter generating sufficient memory pressure to create noticeable contention, which the receiver must detect. Essentially, the transmitter needs to generate enough memory access requests to contend with the receiver, and the receiver must be sensitive enough to observe the difference between the transmitter's sleep and data copy operations. To evaluate this, we conducted experiments on an Orin Nano using three CPU cores for both the transmitter and receiver. We send 1024 bits of information, evenly distributed with bits '0's and '1's, with the slowdown results illustrated in Figure 6.5.a. We design varying buffer sizes of data copy for transmitter (from 0.6 MB to 128 MB) and receiver (from 1 MB to 100 MB). Overall, we observe an increasing pattern of average slowdown on the receiver side as we increase transmitter buffer sizes. For example, with a 1 MB buffer size for the receiver, the transmitter requires at least a 2 MB buffer (*i.e.*, at least 20% MC utilization) to achieve a minimum of 10% slowdown.

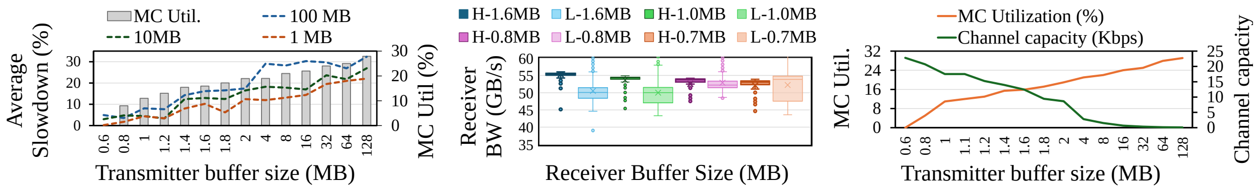


Figure 6.5 (a) [Left] Average slowdown in the perceived BW depending on the transmitter buffer size. (b) [Middle] Average perceived high BW (H) and low BW (L) for bits '0' and '1', respectively, for varying receiver buffer sizes. (c) [Right] MC utilization per transmitter buffer size.

Although we observe a slowdown while transmitting bit '0', we also need to clearly distinguish the BW differences between bit '1' and bit '0' on the receiver. To demonstrate this, we measure and analyze the average BW perceived on the receiver side while the transmitter is running, with a 4 MB buffer size sending '0' (higher perceived BW) and '1' (lower perceived BW) bits. Figure 6.5.b depicts the distribution (including outliers) of the BW perceived by the receiver while sensing '0's (light colors) and '1's (dark colors). Although buffer sizes of 1.6 MB and 1.0 MB have clear differences in terms of perceived BW, lower buffer sizes for the receiver fail to distinguish the differences between bit '0' and '1' by looking at perceived BW. While outliers create noise if accuracy is calculated solely with average-based methods, history-based methods (which compare the current trace with the previous) clearly identify the changes. Although the transmitter with 0.8 MB buffer has approximately 10% MC utilization, contention may not be enough to distinguish bit '0's and '1's.

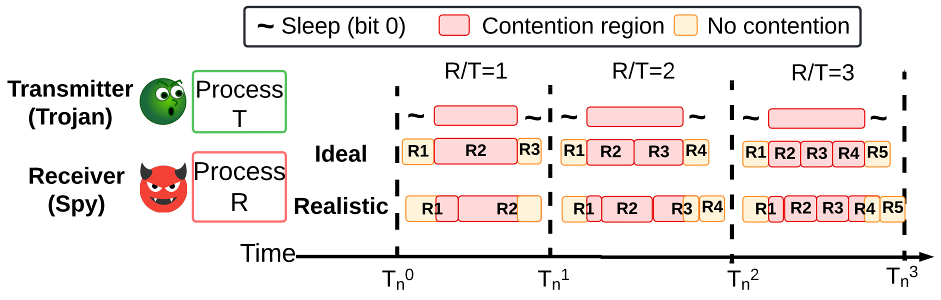


Figure 6.6 The overlap between [T]ransmitter's and [R]eceiver's copy operations for various R/T copy epoch ratios. 'Ideal' represents the expected durations and 'actual' represents the observed. R1-R5 indicates the epoch number of a copy operation performed by the receiver.

6.3.7 Trade-off between Copy Duration and Contention Amount

The channel capacity intuitively depends on the size of the transmitter's buffer that is being copied. Assuming that, to transmit bit '1', transmitter copy operation with a fixed buffer size will be performed once during time interval T_n, there exists an inverse relationship between the transmitter buffer size and the channel capacity, as demonstrated in Eq. 6.1. As we increase the transmitter buffer size, thus the time $Time_{Tra}$ to copy the buffer, and account for the average slowdown $Slowdown_{Rec}$ perceived by the receiver, the channel capacity decreases.

$$Channel\ Capacity = 1\ /\ (Time_{Tra} * Slowdown_{Rec}) \tag{6.1}$$

Ideally, we aim to use the smallest possible transmitter buffer size to maximize the channel capacity of our covert channel. On the other hand, there is a lower limit to how much we can decrease the buffer size to generate observable contention. To demonstrate the trade-off between channel capacity and buffer size, we vary the transmitter buffer size and report the results in Figure 6.5.c. We observe that increasing the transmitter buffer size leads to a near-linear decrease in channel capacity. For example, with transmitter buffer sizes of 0.5 MB and 0.6 MB, we achieved channel capacities of up to 25 Kbps, yet the receiver was unable to detect the transmitter's activity since the transmitter's MC utilization was nearly zero.

6.3.8 Reducing Noise

Synchronizing transmitter and receiver is essential for accurately sensing BW differences. Since direct communication between the transmitter and receiver is not allowed (which would otherwise defeat the point of a covert channel), we must statically decide the time intervals. However, as depicted in Figure 6.5.b, due to contention generation being inherently noisy, the accumulation of outlier-induced errors can lead to desynchronization. If we aim to operate in the worst-case scenario, then many contended regions may complete earlier than anticipated.

We illustrate and compare the expected (*i.e.*, ideal) and observed (*i.e.*, actual) copy operation durations in Figure 6.6. R/T denotes the ratio of copy epochs (or iterations) that the (R)eceiver performs for every bit '1' sent by the (T)ransmitter. When $R/T = 1$, the receiver perceives the contention from the transmitter with a delay, causing a drop in observed BW and increasing in noise Conversely, when the transmitter sends bit '1' across multiple epochs (*i.e.*, $R/T > 1$), the receiver will capture at least one or more fully contended regions. This substantially improves transmission accuracy, while reducing the communication capacity of the covert channel.

6.4 Improving Channel Capacity with GPUs

Mobile and autonomous SoCs often embed GPUs which are designed to enable massive parallelism. This results in better utilization of the memory subsystem compared to the CPUs.

6.4.1 Receiver on the GPU

The receiver must generate sufficiently high BW to accurately distinguish between a '0' bit and a '1' bit. To achieve better accuracy, we run the receiver on the GPU and the transmitter on the CPU. We implement the memory copy operation with `cudaMemcpy` using CUDA [238]. Figure 6.7.a shows the average receiver slowdown for the '0' (*i.e.*, high) and '1' (*i.e.*, low) bits using a buffer size of 1 MB on an Orin AGX. For receiver buffer sizes from 2MB to 8MB, we observe a clear distinction between '0's and '1's. However, buffer sizes of 0.5MB and 1MB fail to sense the contention, whereas buffer sizes of 16MB and beyond may result in a misalignment of the copy epochs as explained in Sec. 6.3.8. We also experimented with configuring the transmitter on the GPU and the receiver on the CPU. However, unlike the case where the receiver is on the GPU, placing the transmitter on the GPU did not improve the distinction of '1' and '0' bits. These results were omitted because of space limitations.

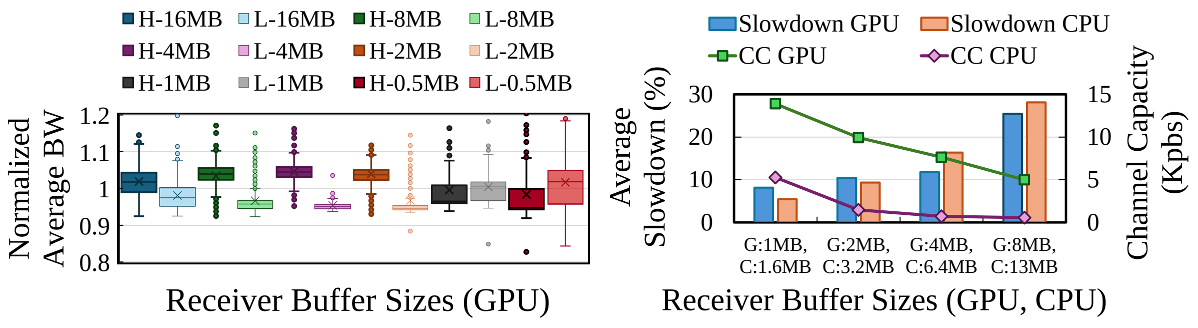


Figure 6.7 (a) [Left] The distribution of the perceived BW distribution for varying buffer sizes. H and L indicates '0' and '1' transmissions, respectively. (b) [Right] Slowdown in the perceived BW and channel capacity when receiver is on (C)PU and (G)PU.

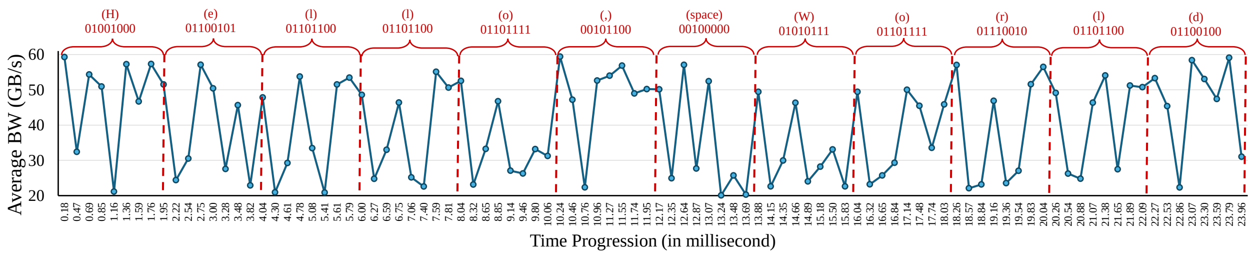


Figure 6.8 Average BW observed by the receiver while receiving a "Hello, World" message.

6.4.2 Channel Capacities for Receivers on GPU and CPU

To further understand the relationship between channel capacity and the slowdown observed when the receiver is placed on CPU and GPU, we perform an experiment on Orin AGX where we gradually increase receiver buffer sizes and report the results in Figure 6.7.b. We observe that we can achieve channel capacities of up to 14 Kbps and 5 Kbps on GPU and CPU, respectively, with 9% and 7% slowdowns observed in the measured BW. As we increase buffer sizes, we typically observe less channel capacity but more sensible contention on both CPU- and GPU-based receivers. It is worth noting that, when the receiver runs on GPU, we can map the transmitter to 11 cores out of the 12 available CPU cores. This mapping significantly increases the contention generation capacity of the transmitter. Overall, the GPU-based receiver achieves approximately $3\times - 5\times$ higher channel capacities on Orin AGX (and $2\times - 4\times$ on Orin Nano) compared to the CPU-based receiver.

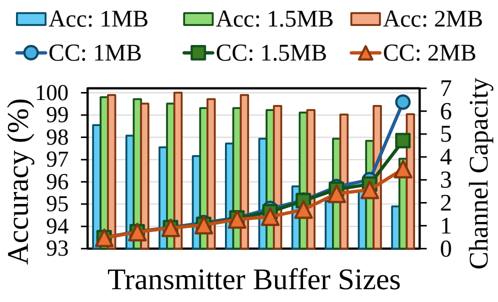
6.4.3 "Hello World" Transmission

To assess our design's performance with longer messages, we transmit a 100 Kb text message on Orin Nano and observe how the perceived BW changes over time. The results are depicted in Figure 6.8. The y-axis shows the rolling average of perceived BW during each time interval, whereas the x-axis represents the time progression in milliseconds. The initial portion of the message is transmitted and received with 100% accuracy while the entire message is delivered with 99.02% accuracy at a channel capacity in excess of 4 Kbps.

6.4.4 Channel Capacity vs. Transmission Accuracy

As the final overarching experiment, we vary the transmitter and buffer sizes and observe the resulting trade-off between channel capacity and transmission accuracy when the receiver is run on the GPU and the transmitter on the CPU of Orin AGX. In Figure 6.9.a, we varied the transmitter buffer sizes while keeping the receiver buffer size fixed at 1 MB. We increase the channel capacity by varying the number of copy epochs/iterations of the buffer copy operation per time interval from 1 to 10, and adjusting the receiver accordingly. In general, our results demonstrate that MC^3 achieves either up to 6.4 Kbps channel capacity or up to 99.99% transmission accuracy. As the transmitter buffer size and the number of copy iterations per interval increased, we observe higher accuracy but reduced channel capacity. Some notable data points are:

- The 2MB transmitter buffer size achieves 99. 1% accuracy at 3.5 Kpbs channel capacity and a near-perfect accuracy of 99.99% at 1.3 Kpbs.
- The 1MB transmitter buffer size maximizes channel capacity up to 6.4 Kpbs while achieving a decent 94.9% accuracy.



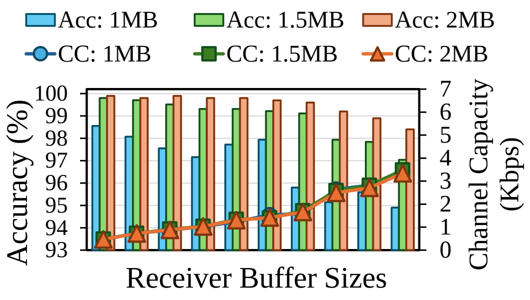


Figure 6.9 The trade-off between accuracy and channel capacity for varying (a) transmitter [left] and (b) receiver buffer sizes [right].

In Figure 6.9.b, we increase receiver's buffer size and buffer copy operation iterations per interval, but keep the transmitter buffer size constant at 1 MB. Overall, increasing the receiver buffer size (with a constant transmitter copy iteration) improved accuracy with minimal impact on channel the capacity. It is worth noting that increasing the receiver size degrades the accuracy since R/T ratio becomes unbalanced once the buffer size is 5 MB and beyond. Similar to the observations in Figure 6.9.a, increasing the channel capacity by decreasing the transmitter copy operations per interval leads to a decrease in accuracy.

6.5 Conclusion

In conclusion, we demonstrate a novel and efficient covert channel attack that exploits shared-memory contention in SM-SoCs. The proposed attack does not require privileged access to the system and achieves a channel bandwidth of up to 6.4 Kbps with accuracy rates that reach 99.99%. We unveil an important vulnerability that could be used to leak private data in modern mobile and autonomous systems.

CHAPTER 7
CONCLUSION AND FUTURE WORK

In this chapter, we will present the conclusion and will then discuss some ideas with possible directions. There are three directions for future work. First, extending the capabilities of resource limited devices to efficiently run modern AI workloads requiring a vast amount of memory and computation capability. Second, resource management and scheduling methodologies we have developed in Chapter 3, 4, and 5 can be extended to have context-awareness for LLM-powered autonomous agents. Third, the covert-channel countermeasures introduced in Chapter 6 should be broadened and hardened to restore end-to-end security as new attack surfaces emerge.

7.1 Conclusion

Edge computing promises to bring low latency, energy efficient intelligence to resource constrained platforms by collaboratively utilizing multiple accelerators within an edge device and across edge and cloud systems. We explored and investigated further tradeoff between performance, power, and security in such heterogeneous environments by solver based schedulers, and integrating lightweight analytical and empirical models.

To address the critical need for energy efficiency in battery-powered edge devices, we introduced *AxoNN*, an energy-aware scheduling framework tailored for single DNN inference tasks on multi-accelerator SoCs. *AxoNN* pioneers an approach that leverages constraint-based optimization to facilitate effective energy-performance trade-offs. By meticulously characterizing the energy and performance profiles of DNN layers across different accelerators and modeling the costs associated with inter-accelerator data transitions, *AxoNN* achieves remarkable prediction accuracy—98% for time and 97% for energy—allowing systems to meet performance goals within specified energy budgets. By enabling precise energy-performance trade-offs via principled optimization, *AxoNN* provides a critical contribution towards sustainable and high-performance AI on energy-limited edge systems. This contribution provides a vital tool for developers seeking to balance the often-conflicting demands of low latency and low power consumption in edge AI applications.

Building upon the insights gained from single-DNN scheduling, we developed *HaX-CONN* to tackle the complexities of concurrently executing multiple DNNs on shared memory SoCs. Recognizing that shared memory contention is a primary performance bottleneck in such scenarios, *HaX-CONN* incorporates a novel, contention-aware runtime. It models per-layer characteristics, shared memory contention effects, and inter-accelerator transition overheads. By formulating the scheduling problem as a satisfiability-modulo-theory (SMT) instance and employing SAT solvers, *HaX-CONN* generates optimal schedules that significantly outperform existing approaches, demonstrating up to a 32% reduction in latency and a 29% improvement in throughput. *HaX-CONN* redefines concurrent AI scheduling on multi-accelerator SoCs by using formal methods to mitigate shared memory contention, thereby boosting system throughput and utilization. Furthermore, its dynamic adaptation capabilities ensure that performance gains are maintained even as workloads change, making it a robust solution for real-world edge environments.

Expanding our focus from single-device optimization to distributed systems, we introduced *HARNESS*, a holistic and scalable resource management framework designed for diversely scaled edge-cloud systems. *HARNESS* addresses the challenge of managing resources across multiple, heterogeneous tiers, often operating under segregated or isolated conditions. It employs a hierarchical graph abstraction (HW-GRAPH) to capture the complex, multi-tier heterogeneity both within and among devices, coupled with a multi-tiered orchestration mechanism. This novel approach allows for decentralized yet coordinated resource management, enabling HARNESS to improve end-to-end latency by up to 47% and drastically reduce performance prediction error rates from 27.4% to just 3.2%. *HARNESS* contributes a novel, decentralized orchestration mechanism, enabling efficient and scalable resource management critical for robust, high-performance edge-cloud applications. *HARNESS* provides a blueprint for managing the next generation of distributed edge-cloud applications effectively.

Finally, acknowledging that enhanced connectivity and shared resources can introduce new security risks, we investigated the security vulnerabilities of shared memory SoCs. This investigation led to the discovery and development of MC^3 , a high-bandwidth covert channel attack that exploits contention in shared DRAM. Unlike previous attacks that often relied on shared caches or privileged access, MC^3 operates across accelerators without elevated privileges, demonstrating the ability to achieve kilobit-per-second transmission rates with less than a 1% error rate on modern edge devices. By unveiling this critical vulnerability, MC^3 underscores the inadequacy of current security measures and highlights the urgent need for new defenses, particularly focusing on DRAM arbitration and memory-centric side channels in heterogeneous systems.

Collectively, the contributions presented in this thesis represent a significant step forward in the field of heterogeneous edge computing. We have demonstrated that through constraint-based optimization and formal methods, it is possible to achieve near-optimal task scheduling for both single and multiple DNN workloads, balancing energy and performance with high precision. We have introduced a scalable hardware graph abstraction and a decentralized orchestration mechanism, proving that efficient, QoS-driven resource management is achievable in complex, multi-tiered edge-cloud systems without requiring a monolithic controller. Furthermore, we have exposed a critical security vulnerability, shifting attention towards the security implications of shared memory design in multi-accelerator platforms. These methodologies and tools, spanning energy-aware scheduling, high-throughput concurrent execution, holistic distributed resource management, and security analysis, provide a cohesive and principled foundation for optimizing the performance, energy consumption, scalability, and security of current and future heterogeneous computing systems on the edge. This work not only addresses pressing contemporary challenges but also lays the groundwork for future innovations in this rapidly evolving and critically important field.

7.2 Future Work

In this subsection, we list three potential directions for future research. First, the deployment CFG-guided weight paging can alleviate edge-device LLM memory pressure and contention. Then, building context-aware scheduling that alternates LLM and heuristic control should balance autonomy and energy. Last, a methodology on countermeasurements should rigorously test heterogeneous DRAM channels and implement QoS defences against emerging covert attacks.

Efficient scheduling for emerging workloads on SoCs: LLM running on edge devices frequently operates under severe memory pressure because a single large-footprint LLM must coexist with latency-critical sensor, vision, and networking tasks. Rather than shrinking the model or resorting to aggressive low-precision quantization, which both compromise accuracy, an application-aware, user-level paging system that treats the model's weights as a first-class, schedulable data set. The key insight is that the order in which a transformer traverses its layers and KV-cache is largely deterministic and can be captured in a lightweight control-flow graph (CFG) extracted at compile time. By analyzing this CFG offline and updating it with low-cost online profiling, the runtime can precisely pre-stage the subtensors that will be referenced in the next execution window. If the system bandwidth allows and is capable of moving data efficiently, serving them directly from flash to the compute engine while bypassing DRAM altogether would increase the performance. This approach will require a careful analysis and strategic engineering solution. First, the system must be designed to operate under varying characteristics of user-triggered queries and fluctuating DRAM footprints from co-resident workloads. A CFG-guided scheduler with a simple admission-control heuristic that monitors DRAM occupancy is very much needed. If the available memory space drops below a safety threshold, newly arriving queries can be temporarily routed down a low-priority path that runs more layers in SSD-resident mode. Future engineering efforts may focus on fine-grained modeling of partial-weight reuse patterns, predictive pacing to hide SSD access variance, and adaptive throttling to mitigate the contention on shared DRAM channels as other applications ramp up or down.

Context-aware resource management: The integration of LLMs into autonomous agents across domains presents significant challenges due to computing and energy constraints and the need for real-time responsiveness. Developing a unified framework that provides not only traditional resource management for the computation component but also context-aware for LLM-powered autonomous systems is needed. A framework can harmonize the computational needs of critical workloads of the system by adapting to the environmental context and system requirements. By collecting and analyzing data from various sensors, the system can understand the current operational environment and context, and then predict resource demands accordingly. Advanced scheduling mechanisms can allocate resources based on task priority, deadlines, and contextual urgency, ensuring that time-sensitive control tasks receive the necessary computational resources. The system can also enable adaptive processing strategies, allowing it to switch between LLM processing and simpler decision-making algorithms depending on context, conserving resources when high-level language processing is unnecessary. Energy efficiency can be optimized by monitoring consumption and adjusting processing loads to prolong battery life without compromising critical functions. This framework can facilitate the deployment of sophisticated LLM capabilities in a wider range of devices and platforms, enhancing the operational effectiveness of autonomous agents in critical applications by ensuring reliable performance and promoting innovation in fields where both advanced AI interactions and real-time control are essential.

Countermeasurements for covert channel attacks: The continuous evolution of side- and covert-channel attacks to circumvent security countermeasures necessitates the continuous development of more sophisticated defense mechanisms. Heterogeneity on phones and edge devices gets more popular: GPU, NPU, ISP, and secure enclaves all share system memory. Future work should test engine-to-CPU and engine-to-engine channels where the accelerators run concurrently. Our covert channel attack, MC^3, unveiled that the shared memory can be a channel for such attacks. While MC^3 proves that memory-contention channels can achieve kilobit-per-second throughput on edge devices (*i.e.*, Jetson devices), mobile phones introduce a harsher and more dynamic operating environment. Parallel defense research can explore bank-level QoS policing, periodic DRAM row randomization, and machine-learning-based detectors that flag persistent single-row hammering patterns while tolerating benign local-buffer accesses. Researchers can both stress-test the security of next-generation mobile SoCs and inspire principled and low-overhead defenses by tackling background noise, adopting low-pressure contention primitives, and circumventing OS scheduling barriers.

REFERENCES

[1] Jeffrey S Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, et al. Extreme heterogeneity 2018-productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity. Technical report, USDOE Office of Science (SC), Washington, DC (United States), 2018.

[2] Baolin Li, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Kairos: Building cost-efficient machine learning inference systems with heterogeneous cloud resources. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 3–16, 2023.

[3] Brice Goglin. Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 74–81. IEEE, 2014.

[4] Mihaela-Andreea Vasile, Florin Pop, Radu-Ioan Tutueanu, Valentin Cristea, and Joanna Kołodziej. Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. *Future Generation Computer Systems*, 51:61–71, 2015.

[5] NVIDIA Deep Learning Accelerator, 2023. URL http://nvdla.org/. (Accessed on 05/01/2025).

[6] Qualcomm Hexagon Digital Signal Processor, 2023. URL https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor. (Accessed on 05/01/2025).

[7] Xilinx. Adaptive compute acceleration platforms. https://www.xilinx.com/products/silicon-devices/acap/versal.html, 2023. (Accessed on 05/01/2025).

[8] MobilEye. Eyeq | the system-on-chip for automotive applications. https://www.mobileye.com/technology/eyeq-chip/, 2023. (Accessed on 05/01/2025).

[9] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey of machine learning accelerators. In *2020 IEEE high performance extreme computing conference (HPEC)*, pages 1–12. IEEE, 2020.

[10] Kun Cao, Shiyan Hu, Yang Shi, Armando Walter Colombo, Stamatis Karnouskos, and Xin Li. A survey on edge and edge-cloud computing assisted cyber-physical systems. *IEEE Transactions on Industrial Informatics*, 17(11):7806–7819, 2021.

[11] Justin McGowen, Ismet Dagli, Neil T Dantam, and Mehmet E Belviranli. Scheduling for cyber-physical systems with heterogeneous processing units under real-world constraints. In *Proceedings of the 38th ACM International Conference on Supercomputing*, pages 298–311, 2024.

[12] Peter Arthurs, Lee Gillam, Paul Krause, Ning Wang, Kaushik Halder, and Alexandros Mouzakitis. A taxonomy and survey of edge cloud computing for intelligent transportation systems and connected vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[13] Mohammad Alaul Haque Monil, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Allen D Malony. Mephesto: Modeling energy-performance in heterogeneous socs and their trade-offs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, pages 413–425, 2020.

[14] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. Cohort: Software-oriented acceleration for heterogeneous socs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 105–117, 2023.

[15] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P Carloni. Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous socs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 350–365, 2021.

[16] Hao Luan, Yu Yao, and Chang Huang. A many-ported and shared memory architecture for high-performance adas socs. *IEEE Design & Test*, 39(6):5–15, 2022.

[17] Mingli Xie, Dong Tong, Kan Huang, and Xu Cheng. Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 344–355. IEEE, 2014.

[18] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou. Shared memory multiplexing: a novel way to improve gpgpu throughput. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, 2012.

[19] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications*, 26(4):178–183, 2019.

[20] Daniel Rosendo, Alexandru Costan, Patrick Valduriez, and Gabriel Antoniu. Distributed intelligence on the edge-to-cloud continuum: A systematic literature review. *Journal of Parallel and Distributed Computing*, 166:71–94, 2022.

[21] Luhui Wang, Cong Zhao, Shusen Yang, Xinyu Yang, and Julie McCann. Ace: Toward application-centric, edge-cloud, collaborative intelligence. *Communications of the ACM*, 66(1):62–73, 2023.

[22] Wuyang Zhang, Sugang Li, Luyang Liu, Zhenhua Jia, Yanyong Zhang, and Dipankar Raychaudhuri. Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1270–1278. IEEE, 2019.

[23] Ang Li, Jingwei Sun, Pengcheng Li, Yu Pu, Hai Li, and Yiran Chen. Hermes: an efficient federated learning framework for heterogeneous mobile clients. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 420–437, 2021.

[24] Young Geun Kim and Carole-Jean Wu. Autofl: Enabling heterogeneity-aware energy efficient federated learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–198, 2021.

[25] Zirui Xu, Fuxun Yu, Jinjun Xiong, and Xiang Chen. Helios: Heterogeneity-aware federated learning with dynamically balanced collaboration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 997–1002. IEEE, 2021.

[26] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Ninad Jadhav, Aleksandra Faust, and Vijay Janapa Reddi. Roofline model for uavs: A bottleneck analysis tool for onboard compute characterization of autonomous unmanned aerial vehicles. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 162–174. IEEE, 2022.

[27] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018.

[28] Behzad Boroujerdian, Ying Jing, Devashree Tripathy, Amit Kumar, Lavanya Subramanian, Luke Yen, Vincent Lee, Vivek Venkatesan, Amit Jindal, Robert Shearer, and Vijay Janapa Reddi. Farsi: An early-stage design space exploration framework to tame the domain-specific system-on-chip complexity. *ACM Transactions on Embedded Computing Systems (TECS)*, 22(2):1–35, 2023.

[29] Xueshi Hou, Yao Lu, and Sujit Dey. Wireless vr/ar with edge/cloud computing. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8. IEEE, 2017.

[30] Shu Shi, Varun Gupta, Michael Hwang, and Rittwik Jana. Mobile vr on edge cloud: a latency-driven design. In *Proceedings of the 10th ACM multimedia systems conference*, pages 222–231, 2019.

[31] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–768, 2019.

[32] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566. IEEE, 2021.

[33] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. Hasco: Towards agile hardware and software co-design for tensor computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1055–1068. IEEE, 2021.

[34] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 874–887, 2022.

[35] Anish Krishnakumar, Umit Ogras, Radu Marculescu, Mike Kishinevsky, and Trevor Mudge. Domain-specific architectures: Research problems and promising approaches. *ACM Transactions on Embedded Computing Systems (TECS)*, 22(2):1–26, 2023.

[36] Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. Democratizing domain-specific computing. *Communications of the ACM*, 66(1):74–85, 2022.

[37] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2021.

[38] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E Belviranli. Axonn: Energy-aware execution of neural network inference on multi-accelerator heterogeneous socs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1069–1074, 2022.

[39] Ismet Dagli and Mehmet E Belviranli. Shared memory-contention-aware concurrent dnn execution for diversely heterogeneous system-on-chips. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP'24)*, pages 243–256, 2024.

[40] Ismet Dagli, Justin Davis, and Mehmet E Belviranli. Harness: Holistic resource management for diversely scaled edge cloud systems. In *Proceedings of the 39th ACM International Conference on Supercomputing*, 2025.

[41] Ismet Dagli, James Crea, Soner Seckiner, Yuanchao Xu, Selçuk Köse, and Mehmet E. Belviranli. MC^3: Memory contention-based covert channel communication on shared dram system-on-chips. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, 2025. doi: 10.23919/DATE64628.2025.10992919.

[42] Yanping Huang. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NIPS*, 2019.

[43] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *SOSP*, 2019.

[44] Jay H. Park. Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism. In *USENIX ATC*, 2020.

[45] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *OSDI*, 2020.

[46] Elham Shamsa, Anil Kanduri, Amir M. Rahmani, Pasi Liljeberg, Axel Jantsch, and Nikil Dutt. Goal-driven autonomy for efficient on-chip resource management: Transforming objectives to goals. In *DATE*, 2019.

[47] Yuanchao Xu, Mehmet Esat Belviranli, Xipeng Shen, and Jeffrey Vetter. Pccs: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips. In *MICRO-54*, 2021.

[48] Hsin-I Wu, Da-Yi Guo, Hsu-Hsun Chin, and Ren-Song Tsay. A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems. In *AICAS*, 2020.

[49] Duseok Kang, Jinwoo Oh, Jongwoo Choi, Youngmin Yi, and Soonhoi Ha. Scheduling of deep learning applications onto heterogeneous processors in an embedded device. In *IEEE Access*, 2020.

[50] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. Deep learning inference parallelization on heterogeneous processors with tensorrt. In *IEEE Embedded Systems Letters*, 2021.

[51] Svetlana Minakova, Erqian Tang, and Todor Stefanov. Combining task-and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings 20*, pages 18–35. Springer, 2020.

[52] Pirah Noor Soomro, Mustafa Abduljabbar, Jeronimo Castrillon, and Miquel Pericàs. An online guided tuning approach to run cnn pipelines on edge devices. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, CF '21, page 45–53, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384049. doi: 10.1145/3457388.3458662. URL https://doi.org/10.1145/3457388.3458662.

[53] Mohammad Alaul Haque Monil, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Allen D. Malony. Mephesto: Modeling energy-performance in heterogeneous socs and their trade-offs. In *PACT*, 2020.

[54] Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. A black-box approach to energy-aware scheduling on integrated cpu-gpu systems. In *CGO*, 2016.

[55] Stavros Tzilis, Pedro Trancoso, and Ioannis Sourdis. Energy-efficient runtime management of heterogeneous multicores using online projection. *TACO*, 2019.

[56] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous dataflow accelerators for multi-dnn workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–83. IEEE, 2021.

[57] Xinyi Zhang, Cong Hao, Peipei Zhou, Alex Jones, and Jingtong Hu. H2h: Heterogeneous model to heterogeneous system mapping with computation and communication awareness. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, 2022.

[58] Seah Kim, Hasan Genc, Vadim Vadimovich Nikiforov, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. Moca: Memory-centric, adaptive execution for multi-tenant deep neural networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 828–841. IEEE, 2023.

[59] Andreas Karatzas and Iraklis Anagnostopoulos. Omniboost: Boosting throughput of heterogeneous embedded devices under multi-dnn workload. *arXiv preprint arXiv:2307.03290*, 2023.

[60] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. {Heterogeneity-Aware} cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[61] Ismet Dagli and Mehmet E Belviranli. Multi-accelerator neural network inference in diversely heterogeneous embedded systems. In *2021 IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop (RSDHA)*, pages 1–7, 2021. doi: 10.1109/RSDHA54838.2021.00006.

[62] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture(ISCA)*, pages 92–104, 2015.

[63] Sheng-Chun Kao and Tushar Krishna. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2020.

[64] Duseok Kang, Jinwoo Oh, Jongwoo Choi, Youngmin Yi, and Soonhoi Ha. Scheduling of deep learning applications onto heterogeneous processors in an embedded device. *IEEE Access*, 8: 43980–43991, 2020.

[65] Hsin-I Wu, Da-Yi Guo, Hsu-Hsun Chin, and Ren-Song Tsay. A pipeline-based scheduler for optimizing latency of convolution neural network inference over heterogeneous multicore systems. In *IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 46–49. IEEE, 2020.

[66] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 159–172. IEEE, 2021.

[67] Stavros Tzilis, Pedro Trancoso, and Ioannis Sourdis. Energy-efficient runtime management of heterogeneous multicores using online projection. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):1–26, 2019.

[68] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. Deep learning inference parallelization on heterogeneous processors with tensorrt. *IEEE Embedded Systems Letters*, 14(1): 15–18, 2021.

[69] Mehmet E. Belviranli, Farzad Khorasani, Laxmi N. Bhuyan, and Rajiv Gupta. Cumas: Data transfer aware multi-application scheduling for shared gpus. In *International Conference on Supercomputing (ICS)*, page 31. ACM, 2016.

[70] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles(SOSP)*, pages 1–15, 2019.

[71] Jay H Park, Gyeongchan Yun, Chang M Yi, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 307–321, 2020.

[72] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479, 2020.

[73] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. Dragon: Breaking gpu memory capacity limits with direct nvm access. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, 2018.

[74] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 27–42, 2022.

[75] Fareed Qararyah, Mohamed Wahib, Doğa Dikbayır, Mehmet Esat Belviranli, and Didem Unat. A computational-graph partitioning method for training memory-constrained dnns. *Parallel computing*, 104:102792, 2021.

[76] Georgios Zacharopoulos, Adel Ejjeh, Ying Jing, En-Yu Yang, Tianyu Jia, Iulian Brumar, Jeremy Intan, Muhammad Huzaifa, Sarita Adve, Vikram Adve, Gu-Yeon Wei, and David Brooks. Trireme: Exploration of hierarchical multi-level parallelism for hardware acceleration. *ACM Transactions on Embedded Computing Systems (TECS)*, 22(3):1–23, 2023.

[77] Justin Davis and Mehmet E Belviranli. Context-aware multi-model object detection for diversely heterogeneous compute systems. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[78] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. *FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System*, page 859–873. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450371025.

[79] Joshua Mack, Sahil Hassan, Nirmal Kumbhare, Miguel Castro Gonzalez, and Ali Akoglu. CEDR: A Compiler-Integrated, Extensible DSSoC Runtime. *ACM Transactions on Embedded Computing Systems (TECS)*, 22(2):1–34, 3 2023. ISSN 1539-9087. doi: 10.1145/3529257.

[80] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S Vetter. Iris: A portable runtime system exploiting multiple heterogeneous programming systems. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2021.

[81] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S Vetter. Iris: A performance-portable framework for cross-platform heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[82] Mehmet E Belviranli, Laxmi N Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–20, 2013.

[83] David Black-Schaffer, Nikos Nikoleris, Erik Hagersten, and David Eklov. Bandwidth bandit: Quantitative characterization of memory contention. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10. IEEE Computer Society, 2013.

[84] Shin-Ying Lee and Carole-Jean Wu. Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 43–53. IEEE, 2017.

[85] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 107–120, 2019.

[86] Amirhossein Mirhosseini and Thomas Wenisch. μsteal: a theory-backed framework for preemptive work and resource stealing in mixed-criticality microservices. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 102–114, 2021.

[87] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. Calciom: Mitigating i/o interference in hpc systems through cross-application coordination. In *2014 IEEE 28th international parallel and distributed processing symposium*, pages 155–164. IEEE, 2014.

[88] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Caliper: Interference estimator for multi-tenant environments sharing architectural resources. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–25, 2019.

[89] Sergey Grizan, David Chu, Alec Wolman, and Roger Wattenhofer. djay: Enabling high-density multi-tenancy for cloud gaming servers with dynamic cost-benefit gpu load balancing. In *Proceedings of the sixth ACM symposium on cloud computing (SoCC)*, pages 58–70, 2015.

[90] Dipanjan Sengupta, Anshuman Goswami, Karsten Schwan, and Krishna Pallavi. Scheduling multi-tenant cloud workloads on accelerator-based systems. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 513–524. IEEE, 2014.

[91] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. A multi-neural network acceleration architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 940–953. IEEE, 2020.

[92] Yuqi Xue, Yiqi Liu, Lifeng Nai, and Jian Huang. V10: Hardware-assisted npu multi-tenancy for improved resource utilization and fairness. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[93] Yuanchao Xu, Mehmet Esat Belviranli, Xipeng Shen, and Jeffrey Vetter. Pccs: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1282–1295, 2021.

[94] Mark Hill and Vijay Janapa Reddi. Gables: A roofline model for mobile socs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 317–330. IEEE, 2019.

[95] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing (SoCC)*, pages 173–189, 2022.

[96] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, 2022.

[97] Mohammad Javad Rashti, Jonathan Green, Pavan Balaji, Ahmad Afsahi, and William Gropp. Multi-core and network aware mpi topology functions. In *Recent Advances in the Message Passing Interface: 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings 18*, pages 50–60. Springer, 2011.

[98] Eddy Caron and Frédéric Desprez. Diet: A scalable toolbox to build network enabled servers on the grid. *The International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[99] R Kanniga Devi and G Murugaboopathi. An efficient clustering and load balancing of distributed cloud data centers using graph theory. *International Journal of Communication Systems*, 32(5): e3896, 2019.

[100] Deepal Jayasinghe, Calton Pu, Tamar Eilam, Malgorzata Steinder, Ian Whally, and Ed Snible. Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement. In *2011 IEEE international conference on services computing (SCC)*, pages 72–79. IEEE, 2011.

[101] Bharath Phanibhushana and Sandip Kundu. Network-on-chip design for heterogeneous multiprocessor system-on-chip. In *2014 IEEE computer society annual symposium on VLSI*, pages 486–491. IEEE, 2014.

[102] Yong Zheng, Haigang Yang, Yi Shu, Yiping Jia, and Zhihong Huang. mtree: A customized multicast-enabled tree-based network on chip for ai chips. *IEEE Embedded Systems Letters (ESL))*, 14(3):143–146, 2022.

[103] Yuzhe Ma, Zhuolun He, Wei Li, Lu Zhang, and Bei Yu. Understanding graphs in eda: From shallow to deep learning. In *Proceedings of the 2020 International Symposium on Physical Design*, pages 119–126, 2020.

[104] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. IEEE, 2010.

[105] Edgar A León, Brice Goglin, and Andres Rubio Proaño. M&mms: navigating complex memory spaces with hwloc. In *Proceedings of the International Symposium on Memory Systems*, pages 149–155, 2019.

[106] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*, pages 863–874. Springer, 2009.

[107] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[108] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216, 2014.

[109] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02):173–193, 2011.

[110] Guangnan Feng, Dezun Dong, Shizhen Zhao, and Yutong Lu. Grap: Group-level resource allocation policy for reconfigurable dragonfly network in hpc. In *Proceedings of the 37th International Conference on Supercomputing (ICS)*, pages 437–449, 2023.

[111] Ivy Peng, Ian Karlin, Maya Gokhale, Kathleen Shoga, Matthew Legendre, and Todd Gamblin. A holistic view of memory utilization on hpc systems: Current and future trends. In *Proceedings of the International Symposium on Memory Systems*, pages 1–11, 2021.

[112] Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. Merchandiser: Data placement on heterogeneous memory for task-parallel hpc applications with load-balance awareness. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 204–217, 2023.

[113] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 119–130, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450335591. doi: 10.1145/2751205.2751213. URL https://doi.org/10.1145/2751205.2751213.

[114] Nikhil Jain, Abhinav Bhatele, Louis H Howell, David Böhme, Ian Karlin, Edgar A León, Misbah Mubarak, Noah Wolfe, Todd Gamblin, and Matthew L Leininger. Predicting the performance impact of different fat-tree configurations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2017.

[115] Nikhil Jain, Abhinav Bhatele, Xiang Ni, Nicholas J Wright, and Laxmikant V Kale. Maximizing throughput on a dragonfly network. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 336–347. IEEE, 2014.

[116] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the tenth european conference on computer systems (EuroSys)*, pages 1–17, 2015.

[117] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Proceedings of the fifteenth European conference on computer systems (EuroSys)*, pages 1–14, 2020.

[118] Kevin A Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen D Malony, Thomas Sterling, and Rob Fowler. An autonomic performance environment for exascale. *Supercomputing frontiers and innovations*, 2(3):49–66, 2015.

[119] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4): 1422–1434, 2018.

[120] Hsin-Hsuan Sung, Jou-An Chen, Wei Niu, Jiexiong Guan, Bin Ren, and Xipeng Shen. Decentralized {Application-Level} adaptive scheduling for {Multi-Instance}{DNNs} on open mobile devices. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 865–877, 2023.

[121] Halima Bouzidi, Mohanad Odema, Hamza Ouarnoughi, Smail Niar, and Mohammad Abdullah Al Faruque. Map-and-conquer: Energy-efficient mapping of dynamic neural nets onto heterogeneous mpsocs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[122] Rathinaraja Jeyaraj, Anandkumar Balasubramaniam, Ajay Kumara MA, Nadra Guizani, and Anand Paul. Resource management in cloud and cloud-influenced technologies for internet of things applications. *ACM Computing Surveys*, 55(12):1–37, 2023.

[123] Yuanming Ren, Shihao Shen, Yanli Ju, Xiaofei Wang, Wenyu Wang, and Victor CM Leung. Edgematrix: A resources redefined edge-cloud system for prioritized services. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 610–619. IEEE, 2022.

[124] Yogesh D Barve, Shashank Shekhar, Ajay Chhokra, Shweta Khare, Anirban Bhattacharjee, Zhuangwei Kang, Hongyang Sun, and Aniruddha Gokhale. Fecbench: A holistic interference-aware approach for application performance modeling. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 211–221. IEEE, 2019.

[125] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy (S&P)*, pages 605–622. IEEE, 2015.

[126] Wu Zhenyu, Xu Zhang, and H Wang. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[127] Qisheng Jiang and Chundong Wang. Sync+ sync: A covert channel built on fsync with storage. *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[128] Jeferson González-Gómez, Kevin Cordero-Zuñiga, Lars Bauer, and Jörg Henkel. The first concept and real-world deployment of a gpu-based thermal covert channel: Attack and countermeasures. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[129] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan Van Schaik, Daniel Genkin, and Yuval Yarom. Hot pixels: Frequency, power, and temperature attacks on {GPUs} and arm {SoCs}. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6275–6292, 2023.

[130] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Dos attacks on your memory in cloud. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (CCS)*, pages 253–265, 2017.

[131] Aditya S Gangwar, Prathamesh N Tanksale, Shirshendu Das, and Sudeepta Mishra. Flush+ early reload: Covert channels attack on shared llc using mshr merging. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[132] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. {ARMageddon}: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, 2016.

[133] Yun Chen, Arash Pashrashid, Yongzheng Wu, and Trevor E Carlson. Prime+ reset: Introducing a novel cross-world covert-channel through comprehensive security analysis on arm trustzone. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[134] Sankha Baran Dutta, Hoda Naghibijouybari, Arjun Gupta, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Spy in the gpu-box: Covert and side channel attacks on multi-gpu systems. In *ISCA*, pages 1–13, 2023.

[135] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems. In *ISCA*, pages 972–984. IEEE, 2021.

[136] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *Financial Cryptography and Data Security: 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers 21*, pages 247–267. Springer, 2017.

[137] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *2016 IEEE symposium on security and privacy (SP)*, pages 987–1004. IEEE, 2016.

[138] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[139] Thales Bandiera Paiva, Javier Navaridas, and Routo Terada. Robust covert channels based on dram power consumption. In *Information Security: 22nd International Conference, ISC 2019, New York City, NY, USA, September 16–18, 2019, Proceedings 22*, pages 319–338. Springer, 2019.

[140] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. {DRAMA}: Exploiting {DRAM} addressing for {Cross-CPU} attacks. In *25th USENIX security symposium (USENIX security 16)*, pages 565–581, 2016.

[141] Yang Luo, Wu Luo, Xiaoning Sun, Qingni Shen, Anbang Ruan, and Zhonghai Wu. Whispers between the containers: High-capacity covert channel attacks in docker. In *2016 IEEE trustcom/bigdatase/ispa*, pages 630–637. IEEE, 2016.

[142] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (ASIA-CCS)*, pages 753–768, 2019.

[143] Thomas Moscibroda Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX security*, 2007.

[144] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3):335–346, 2010.

[145] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010.

[146] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. Avoiding information leakage in the memory controller with fixed service policies. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 89–101, 2015.

[147] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *HPCA*, pages 225–236. IEEE, 2014.

[148] Nikolay Matyunin, Nikolaos A Anagnostopoulos, Spyros Boukoros, Markus Heinrich, André Schaller, Maksim Kolinichenko, and Stefan Katzenbeisser. Tracking private browsing sessions using cpu-based covert channels. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSeC)*, pages 63–74, 2018.

[149] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. {AutoLock}: Why cache attacks on {ARM} are harder than you think. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1075–1091, 2017.

[150] S Karen Khatamifard, Longfei Wang, Amitabh Das, Selçuk Köse, and Ulya R Karpuzcu. Powert channels: A novel class of covert communicationexploiting power management vulnerabilities. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 291–303. IEEE, 2019.

[151] Marvin Damschen, Frank Mueller, and Jörg Henkel. Co-scheduling on fused cpu-gpu architectures with shared last level caches. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2337–2347, 2018. doi: 10.1109/TCAD.2018.2857042.

[152] Sitao Huang. Analysis and modeling of collaborative execution strategies for heterogeneous cpu-fpga architectures. In *ICPE*, 2019.

[153] Svetlana Minakova and Erqian Tang. Combining task- and data-level parallelism for high-throughput cnn inference on embedded cpus-gpus mpsocs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2020.

[154] Sheng-Chun Kao and Tushar Krishna. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *ICCAD*, 2020.

[155] Houssam-Eddine Zahaf, Richard Olejnik, Giuseppe Lipari, et al. Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms. *Journal of Systems Architecture*, 74: 46–60, 2017.

[156] Li Han, Yiqin Gao, Jing Liu, Yves Robert, and Frédéric Vivien. Energy-aware strategies for reliability-oriented real-time task allocation on heterogeneous platforms. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388160. doi: 10.1145/3404397.3404419. URL https://doi.org/10.1145/3404397.3404419.

[157] Hongzhi Xu, Renfa Li, Chen Pan, and Keqin Li. Minimizing energy consumption with reliability goal on heterogeneous embedded systems. In *JPDC*, 2019.

[158] Ismet Dagli and Mehmet E. Belviranli. Multi-accelerator neural network inference in diversely heterogeneous embedded systems. In *RSDHA Workshop*, 2021.

[159] Maria Dávila, Raúl Nozal, Rubén Gran Tejero, María Villarroya, Darío Suárez Gracia, and Jose Bosque. Cooperative cpu, gpu, and fpga heterogeneous execution with enginecl. *The Journal of Supercomputing*, 75, 03 2019. doi: 10.1007/s11227-019-02768-y.

[160] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[161] NVIDIA. Tensorrt, 2023. URL https://developer.nvidia.com/tensorrt. (Accessed on 05/01/2025).

[162] Tianqi Chen. Tvm: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[163] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[164] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.

[165] Christian Szegedy. Going deeper with convolutions. In *CVPR*, 2015.

[166] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Sabrina Neuman, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. Automatic domain-specific soc design for autonomous unmanned aerial vehicles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 300–317. IEEE, 2022.

[167] Zishen Wan, Karthik Swaminathan, Pin-Yu Chen, Nandhini Chandramoorthy, and Arijit Raychowdhury. Analyzing and improving resilience and robustness of autonomous systems. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[168] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[169] Tesla. Tesla autopilot ai, 2023. URL https://www.tesla.com/AI.

[170] Miao Wang, Xu-Quan Lyu, Yi-Jun Li, and Fang-Lue Zhang. Vr content creation and exploration with deep learning: A survey. *Computational Visual Media*, 6:3–28, 2020.

[171] Ismet Dagli, Andrew Depke, Andrew Mueller, Md Sahil Hassan, Ali Akoglu, and Mehmet Esat Belviranli. Contention-aware performance modeling for heterogeneous edge and cloud systems. In *Proceedings of the 3rd Workshop on Flexible Resource and Application Management on the Edge*, FRAME '23, page 27–31, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701641. doi: 10.1145/3589010.3594889. URL https://doi.org/10.1145/3589010.3594889.

[172] Jérémie Guiochet, Mathilde Machin, and Hélène Waeselynck. Safety-critical advanced robots: A survey. *Robotics and Autonomous Systems*, 94:43–52, 2017.

[173] Nathaniel Hudson, Hana Khamfroush, and Daniel E Lucani. Qos-aware placement of deep learning services on the edge with multiple service implementations. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, pages 1–8. IEEE, 2021.

[174] Yuanchao Xu, Mehmet Esat Belviranli, Xipeng Shen, and Jeffrey Vetter. Pccs: Processor-centric contention-aware slowdown model for heterogeneous system-on-chips. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 1282–1295, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480101. URL https://doi.org/10.1145/3466752.3480101.

[175] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations(ICLR)*, 2014.

[176] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016. doi: 10.1109/MICRO.2016.7783725.

[177] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation(PLDI)*, pages 883–898, 2021.

[178] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

[179] Qualcomm. Neural processing sdk for ai, 2023. URL https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk. (Accessed on 05/01/2025).

[180] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 943–958, 2021.

[181] Sheng-Chun Kao and Tushar Krishna. Magma: An optimization framework for mapping multiple dnns on multiple accelerator cores. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 814–830. IEEE, 2022.

[182] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, and Tushar Krishna. Flat: An optimized dataflow for mitigating attention performance bottlenecks. *published in arxiv, will appear in Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[183] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, pages 209–221. Association for Computing Machinery New York, NY, USA, 2022.

[184] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of big Data*, 8:1–74, 2021.

[185] NVIDIA. Tensorrt iprofiler, 2023. URL https://docs.nvidia.com/deeplearning/tensorrt/api/c_api/classnvinfer1_1_1_i_profiler.html. (Accessed on 05/01/2025).

[186] Qi Zhu, Bo Wu, Xipeng Shen, Li Shen, and Zhiying Wang. Co-run scheduling with power cap on integrated cpu-gpu systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 967–977. IEEE, 2017.

[187] NVIDIA Nsight Compute, 2022. URL https://docs.nvidia.com/nsight-compute/NsightCompute/index.html. (Accessed on 05/01/2025).

[188] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[189] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

[190] Roberto Sebastiani and Patrick Trentin. Optimathsat: A tool for optimization modulo theories. In *International conference on computer aided verification*, pages 447–454. Springer, 2015.

[191] Sabino Francesco Roselli, Kristofer Bengtsson, and Knut Åkesson. Smt solvers for job-shop scheduling problems: Models comparison and performance evaluation. In *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, pages 547–552. IEEE, 2018.

[192] Jie Zou, Xiaotian Dai, and John A McDermid. Resilience-aware mixed-criticality dag scheduling on multi-cores for autonomous systems. *ACM SIGAda Ada Letters*, 42(1):81–85, 2022.

[193] Behzad Boroujerdian, Radhika Ghosal, Jonathan Cruz, Brian Plancher, and Vijay Janapa Reddi. Roborun: A robot runtime to exploit spatial heterogeneity. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 829–834. IEEE, 2021.

[194] Hengyu Zhao, Yubo Zhang, Pingfan Meng, Hui Shi, Li Erran Li, Tiancheng Lou, and Jishen Zhao. Safety score: A quantitative approach to guiding safety-aware autonomous vehicle computing system design. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 1479–1485. IEEE, 2020.

[195] Yu-Shun Hsiao, Siva Kumar Sastry Hari, Michał Filipiuk, Timothy Tsai, Michael B Sullivan, Vijay Janapa Reddi, Vasu Singh, and Stephen W Keckler. Zhuyi: perception processing rate estimation for safety in autonomous vehicles. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 289–294, 2022.

[196] NVIDIA. Next-level ai performance for next-gen robotics | nvidia jetson orin agx. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/, 2023. (Accessed on 05/01/2025).

[197] NVIDIA. Ai-powered autonomous machines at scale | nvidia jetson agx xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/, 2023. (Accessed on 05/01/2025).

[198] Qualcomm. Snapdragon 865 mobile hardware development kit. https://stage.developer.qualcomm.com/hardware/snapdragon-865-hdk, 2023. (Accessed on 05/01/2025).

[199] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[200] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[201] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[202] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[203] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*, 2017.

[204] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, 2017.

[205] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[206] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[207] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3213–3223, 2016.

[208] Rihards Novickis, Aleksandrs Levinskis, Roberts Kadikis, Vitalijs Fescenko, and Kaspars Ozols. Functional architecture for autonomous driving and its implementation. In *2020 17th Biennial Baltic Electronics Conference (BEC)*, pages 1–6. IEEE, 2020.

[209] Ratheesh Ravindran, Michael J Santora, and Mohsin M Jamali. Multi-object detection and tracking, based on dnn, for autonomous vehicles: A review. *IEEE Sensors Journal*, 21(5):5668–5677, 2020.

[210] Jeffrey S Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, et al. Extreme heterogeneity 2018-productive computational science in the era of extreme heterogeneity: Report for doe ascr workshop on extreme heterogeneity. Technical report, USDOE Office of Science (SC), Washington, DC (United States), 2018.

[211] Hsin-Hsuan Sung, Yuanchao Xu, Jiexiong Guan, Wei Niu, Bin Ren, Yanzhi Wang, Shaoshan Liu, and Xipeng Shen. Brief industry paper: Enabling level-4 autonomous driving on a single 1k off-the-shelf card. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 297–300. IEEE, 2022.

[212] Thomas Rausch, Alexander Rashed, and Schahram Dustdar. Optimized container scheduling for data-intensive serverless edge computing. *Future Generation Computer Systems (FGCS)*, 114: 259–271, 2021.

[213] Stefan Nastic, Thomas Pusztai, Andrea Morichetta, Víctor Casamayor Pujol, Schahram Dustdar, Deepak Vii, and Ying Xiong. Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 206–216. IEEE, 2021.

[214] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, and Jeremy Kepner. Scalable system scheduling for hpc and big data. *Journal of Parallel and Distributed Computing*, 111:76–92, 2018.

[215] Xiang Sun, Nirwan Ansari, and Ruopeng Wang. Optimizing resource utilization of a data center. *IEEE Communications Surveys & Tutorials*, 18(4):2822–2846, 2016.

[216] Huang-Chen Lee and Kai-Hsiang Ke. Monitoring of large-area iot sensors using a lora wireless mesh network system: Design and evaluation. *IEEE Transactions on Instrumentation and Measurement*, 67(9):2177–2187, 2018.

[217] Theo Lynn, Pierangelo Rosati, Arnaud Lejeune, and Vincent Emeakaroha. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 162–169. IEEE, 2017.

[218] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor CM Leung, and Cheng-Hsin Hsu. A survey on cloud gaming: Future of computer games. *IEEE Access*, 4:7605–7620, 2016.

[219] Meta. Powered by ai: Oculus insight, 2019. URL https://ai.meta.com/blog/powered-by-ai-oculus-insight/. (Accessed on 05/01/2025).

[220] Kyle L. Spafford and Jeffrey S. Vetter. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE, IEEE Computer Society Press. ISBN 978-1-4673-0804-5.

[221] Abbas Mehrabi, Matti Siekkinen, Teemu Kämäräinen, and Antti Yla-Jaaski. Multi-tier cloudvr: Leveraging edge computing in remote rendered virtual reality. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 17(2):1–24, 2021.

[222] Mehmet E. Belviranli and Jeffrey S. Vetter. Flame: Graph-based hardware representations for rapid and precise performance modeling. In *IEEE Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019.

[223] Mikko Kivelä, Alex Arenas, Marc Barthelemy, James P Gleeson, Yamir Moreno, and Mason A Porter. Multilayer networks. *Journal of complex networks*, 2(3):203–271, 2014.

[224] Marcus Ritter, Alexander Geiß, Johannes Wehrstein, Alexandru Calotoiu, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. Noise-resilient empirical performance modeling with deep neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 23–34. IEEE, 2021.

[225] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report 4, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[226] Hang Liu, Fahima Eldarrat, Hanen Alqahtani, Alex Reznik, Xavier De Foy, and Yanyong Zhang. Mobile edge cloud system: Architectures, challenges, and approaches. *IEEE Systems Journal*, 12(3): 2495–2508, 2017.

[227] Tobias Pfandzelter, Aditya Dhakal, Eitan Frachtenberg, Sai Rahul Chalamalasetti, Darel Emmot, Ninad Hogade, Rolando Pablo Hong Enriquez, Gourav Rattihalli, David Bermbach, and Dejan Milojicic. Kernel-as-a-service: A serverless programming model for heterogeneous hardware accelerators. In *Proceedings of the 24th International Middleware Conference*, pages 192–206, 2023.

[228] CDC. Niosh mine and mine worker charts. wwwn.cdc.gov/NIOSH-Mining/MMWC, 2023. (Accessed on 05/01/2025).

[229] NVIDIA. Jetpack sdk, (Accessed on 05/01/2025). URL https://developer.nvidia.com/embedded/jetpack.

[230] NVIDIA. Vpi - vision programming interface documentation, 2024. URL https://docs.nvidia.com/vpi/. (Accessed on 05/01/2025).

[231] Yunyang Xiong, Hanxiao Liu, Suyog Gupta, Berkin Akin, Gabriel Bender, Yongzhe Wang, Pieter-Jan Kindermans, Mingxing Tan, Vikas Singh, and Bo Chen. Mobiledets: Searching for object detection architectures for mobile accelerators. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (CVPR)*, pages 3825–3834, 2021.

[232] Ahmad Jalal, Majid Ali Khan Quaid, and Kibum Kim. A wrist worn acceleration based human motion analysis and classification for ambient smart home system. *Journal of Electrical Engineering & Technology*, 14:1733–1739, 2019.

[233] Apple. Apple unveils the new macbook pro featuring the m3 family of chips, 2023. (Accessed on 05/01/2025).

[234] Qualcomm. Snapdragon mobile platforms. https://www.qualcomm.com/snapdragon/products, 2024.

[235] Mohammad Dashti and Alexandra Fedorova. Analyzing memory management methods on integrated cpu-gpu systems. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 59–69, 2017.

[236] Arm® Architecture Reference Manual for A-profile architecture, 2024. URL https://developer.arm.com/documentation/ddi0487/latest.

[237] Blat Blatnik. The perfect Sleep() function. https://blog.bearcats.nl/perfect-sleep-function/, 2024.

[238] NVIDIA. Cuda samples. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/1_Utilities/bandwidthTest, 2024. (Accessed on 05/01/2025).

APPENDIX
PERMISSION STATEMENT

A.1 Conference Articles in Chapters

Chapters 3-6 are drawn directly from the first-author papers by the author [38–41]. The articles used in chapters 3, 4, and 5 are open-sourced to be used/accessed by everyone under CC license. Particularly, the articles in these chapters are licensed CC BY 4.0, which permits full reproduction in a thesis provided the Version-of-Record DOI is cited. Even though it is published in IEEE proceedings, IEEE states that "IEEE is not the copyright holder of the material" for the article we used in Chapter 6. The article we used in Chapter 6 is included in its accepted manuscript form under IEEE's thesis-reuse policy that allows authors to use their own work with the standard IEEE notice. For all articles from chapter 3 to 6 in this thesis, aside from minor stylistic edits, all reused text, figures, and tables are unchanged.